University of California
Santa Barbara

Automated Black Box Generation of Structured Inputs for Use in Software Testing

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Kyle Thomas Dewey

Committee in charge:

Professor Ben Hardekopf, Chair
Professor Chandra Krintz
Professor Tevfik Bultan

September 2017

The Dissertation of Kyle Thomas Dewey is approved.

Professor Chandra Krintz

Professor Tevfik Bultan

Professor Ben Hardekopf, Committee Chair

June 2017

Automated Black Box Generation of Structured Inputs for Use in Software Testing

by

Kyle Thomas Dewey

Dedicated to Melanie, whose love and support has exceeded far beyond anything I could imagine.

Acknowledgements

In many ways, getting to this point was a team effort, and it would not have been possible if not for the vast amount of help I received along the way. Here I point out specific people who have been instrumental in this process, for whom I am immensely thankful.

- **Ben Hardekopf**: For being an all-around excellent adviser and teaching me how to do research in the first place. Your patience speaks tomes.
- **Tevfik Bultan**: For first exposing me to research in software engineering, and plenty of related critical guidance.
- **Chandra Krintz**: For plenty of crucial guidance early on, and convincing me being here was a good idea.
- **Phill Conrad**: For providing tons of mentoring both for teaching and for CS education research.
- **Ben Wiedermann**: For teaching-related mentoring, along with giving CLP-based testing its first practical application).
- **Vineeth Kashyap**: For being my first line of defense against having no idea what was going on in my first two years.
- **Madhukar Kedlaya**: For being my first line of defense against having no idea what was going on in my third year.
- **Lawton Nichols**: For first convincing me that I might actually have an idea of what's going on.
- **Niko Matsakis**, along with the entire Rust development team: For answering barrage after barrage of questions about Rust. We could not have fuzzed Rust without this help.

- **Robert Rothman**: For being brutally honest with me during my undergrad, and letting me know my place was in Computer Science.

- **Michael Christensen, Mehmet Emre, Miroslav (Mika) Gavrilov**: For being wonderful fellow Ph.D. compatriots. You have made this a fun-filled experience. Case in point: Enter Sandman, Sad but True, Holier Than Thou, The Unforgiven, Wherever I May Roam, Don't Tread on Me, Through the Never, Nothing Else Matters, Of Wolf and Man, The God That Failed, My Friend of Misery, The Struggle Within

- **Jared Roesch, Elena Morozova, Dylan Lynch, Ethan Kuefner, Berkeley Churchill, Davina Zamanzadeh, Dianne Wagner, Ben Campbell**: The undergraduate students whom I worked closely with for multiple quarters, if not years. Beyond your research contributions, you all collectively helped me learn how to advise students.

- **The PL Lab**: All members not already mentioned, past and present.

- **My Parents**: For remaining sane and supportive when their only child suddenly decided to move to California.

I apologize if I missed any names; it is not intentional. My memory resembles that of a goldfish.

Curriculum Vitæ

Kyle Thomas Dewey

Education

2011 - 2017	Ph.D. in Computer Science, University of California, Santa Barbara.
2007 - 2011	M.S. in Bioinformatics, Rochester Institute of Technology, Rochester, NY.
2007 - 2011	B.S. in Bioinformatics, Rochester Institute of Technology, Rochester, NY.

Courses Taught at the University of California, Santa Barbara

Spring 2017	CS 162: Programming Languages
Winter 2017	CS 162: Programming Languages
Summer 2016	CS 56: Advanced Applications Programming (co-instructor)
Winter 2016	CS 64: Computer Organization and Design Logic
Fall 2015	CS 64: Computer Organization and Design Logic
Winter 2015	CS 162: Programming Languages
Summer 2014	CS 24: Problem Solving with Computers II
Summer 2012	CS 16: Problem Solving with Computers I

Courses TA'd at the University of California, Santa Barbara

Spring 2014	CS 162: Programming Languages
Winter 2014	CS 162: Programming Languages
Spring 2013	CS 162: Programming Languages
Winter 2013	CS 162: Programming Languages
Spring 2012	CS 189B: Capstone Project, Part B
Winter 2012	CS 189A: Capstone Project, Part A
Fall 2011	CS 170: Operating Systems

Awards and Professional Service

June 2016	Student Volunteer for PLDI'16
Winter 2014	Outstanding Teaching Assistant for CS 162: Programming Languages

Peer-Reviewed Publications

Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming
Kyle Dewey, Phill Conrad, Michelle Craig, Elena Morozova
Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2017

Fuzzing the Rust Typechecker Using CLP
Kyle Dewey, Jared Roesch, Ben Hardekopf
Conference on Automated Software Engineering (ASE), 2015

Automated Data Structure Generation: Refuting Common Wisdom
Kyle Dewey, Lawton Nichols, Ben Hardekopf
International Conference on Software Engineering (ICSE), 2015

A Parallel Abstract Interpreter for JavaScript
Kyle Dewey, Vineeth Kashyap, Ben Hardekopf
Symposium on Code Generation and Optimization (CGO), 2015

Language Fuzzing Using Constraint Logic Programming
Kyle Dewey, Jared Roesch, Ben Hardekopf
Conference on Automated Software Engineering (ASE), 2014

JSAI: A Static Analysis Platform for JavaScript
Vineeth Kashyap, **Kyle Dewey**, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, Ben Hardekopf
Symposium on Foundations of Software Engineering (FSE), 2014

Abstract

Automated Black Box Generation of Structured Inputs for Use in Software Testing

by

Kyle Thomas Dewey

A common problem in automated software testing is the need to generate many inputs with complex structure in a black-box fashion. For example, a library for manipulating red-black trees may require that inputs are themselves valid red-black trees, meaning anything invalid is not suitable for testing. As another example, in order to test code generation in a compiler, it is necessary to use input programs which are both syntactically valid and well-typed. Despite the importance of this problem, we observe that existing solutions are few in number and have severe drawbacks, including unreasonably slow performance and a lack of generality to testing different systems.

This thesis presents a solution to this problem of black-box structured input generation. I observe that test inputs can be described as solutions to systems of logical constraints, and that more expressive constraints can lead to more complex tests. In order to test effectively and generate many tests, we need high-performance constraint solvers capable of finding many solutions to these constraints. I observe that constraint logic programming (CLP) offers an expressive constraint language paired with a high-performance constraint solver, and thus serves as a potential solution to this problem. Via a series of case studies, I have found that CLP (1) is applicable to testing a wide variety of systems; (2) can scale to more complex constraints than ever previously described; and (3) is often orders of magnitude faster than competing solutions. These case studies have also exposed dozens of bugs in high-profile software, including the Rust compiler and the Z3 SMT solver.

Contents

Curriculum Vitae **vii**

Abstract **ix**

List of Figures **xiii**

List of Tables **xvi**

1 Introduction and Motivation **1**
1.1 What is Black-Box Fuzzing? 2
1.2 Why (Not) Black-Box Fuzzing? 3
1.3 Related Work 4
1.4 Problem with Black-Box Fuzzing: Highly Structured Input Generation 9
1.5 Key Insights 10
1.6 Potential Solutions 14
1.7 Overarching Thesis 20

2 Case Study: Generating Interesting JavaScript Programs **24**
2.1 Introduction 24
2.2 CLP for Program Generation 25
2.3 Generating JavaScript 31
2.4 Evaluation 37
2.5 Conclusions 41

3 Case Study: Generating Complex Data Structures **43**
3.1 Introduction 43
3.2 Example 45
3.3 CLP Compared to Other Data Structure Generators 46
3.4 Data Structures and Properties 49
3.5 Evaluation 55

3.6 Conclusions 62

4 Case Study: Type-Based Fuzzing of the Rust Compiler 63
4.1 Introduction 63
4.2 Generating Well-Typed Programs 64
4.3 Finding Typechecker Bugs 69
4.4 Testing the Rust Typechecker 76
4.5 Evaluation 83
4.6 Conclusions 89

5 Case Study: Semantics-Based Fuzzing of SMT Solvers 90
5.1 Introduction 90
5.2 Generating Satisfiable Formulas 92
5.3 Generating Unsatisfiable Formulas 98
5.4 Application to Bitvectors and Floating Point 101
5.5 Evaluation 105
5.6 Discussion 110
5.7 Conclusions 113

6 Case Study: Intelligent Fuzzing of Student Tokenizers and Parsers 115
6.1 Introduction 115
6.2 Testing Tokenizers, Parsers, and Arithmetic Evaluators With CLP 116
6.3 Student Programming Assignment 122
6.4 Evaluation 124
6.5 Conclusions 131

7 Case Study: Generating Polymorphic Programs for Testing Student Typecheckers 132
7.1 Introduction 132
7.2 SimpleScala Language 133
7.3 A Naive CLP-Based Generator for Well-Typed SimpleScala Programs 140
7.4 Optimizing the Naive CLP-Based Generator for Well-Typed SimpleScala Programs 156
7.5 Results 169
7.6 Conclusion 169

8 Improving CLP for Testing: Typed-Prolog 171
8.1 Introduction and Motivation 171
8.2 Related Work 174
8.3 Problems with CLP for Test Case Generation 175
8.4 Type System 183

8.5 Compiling Higher-Order Relations . . . 189
8.6 Module System . . . 195
8.7 Results and Discussion . . . 198
8.8 Conclusions . . . 201

9 Improving CLP for Testing: Bounding and Search-Oriented Metainterpreter 202
9.1 Introduction and Motivation . . . 202
9.2 Related Work . . . 206
9.3 Background on CLP Metainterpreters . . . 208
9.4 A Metainterpreter for CLP that Parameterizes Search and Bounding . . . 213
9.5 Composing Search and Bounding Strategies . . . 227
9.6 Results and Discussion . . . 234
9.7 Conclusions and Future Work . . . 235

10 Conclusions and Future Work 237

A CLP Preliminaries 238
A.1 CLP Background . . . 238
A.2 A Taste of CLP . . . 239

B Formal Notation Used 261
B.1 Tuples . . . 261
B.2 Lists . . . 261
B.3 Sets . . . 262
B.4 Maps . . . 262
B.5 Axioms, Inference Rules, and Typing Rules . . . 263

C Full System F Generator 266

D Helper Functions in Typed-Prolog 271

E Helper Functions in SimpleScala 276

Bibliography 285

List of Figures

1.1 Simplistic grammar handling expressions with nats and bools 11
1.2 Encoding of the grammar in Figure 1.1 into equivalent logical inference rules. 11

2.1 Basic arithmetic expression language, consisting of integers and addition over expressions. 25
2.2 CLP implementation of the grammar shown in Figure 2.1 26
2.3 CLP code augmented with stochastic capabilities 29
2.4 CLP code snippet which can generate programs exhibiting arithmetic overflow 30
2.5 A fragment of an unsound type system for JavaScript to filter out programs that attempt to directly access a property of the `null` value. 33
2.6 CLP code implementing the type system fragment shown in Figure 2.5. 33
2.7 CLP code implementing a generator of JavaScript expressions which use prototype-based inheritance 35
2.8 CLP code implementing a generator of JavaScript expressions which use `with` combined with closure creation 36

3.1 CLP-based generator of sorted linked lists. 45

4.1 Syntax for System F 66
4.2 Typing rules for System F 66
4.3 CLP specification of System F 67
4.4 CLP specification of *almost* well-typed System F 72
4.5 Snippet of sanitized code handling two of the three possible Rust type constraints we consider, which is used as a constraint solver 81

5.1 Syntax for the subset of the theory of bitvectors we consider 93
5.2 Type system for the subset of the theory of bitvectors defined in Figure 5.1 93
5.3 Formal big-step semantics for a subset of the theory of bitvectors 94
5.4 CLP implementation of the rules defined in Figure 5.3 96

5.5 Example showing some of the issues involved in generating guaranteed unsatisfiable formulas 99

6.1 Small grammar used for running CLP example 117
6.2 CLP-based tokenizer for the language defined in Figure 6.1 118
6.3 CLP-based parser for the language defined in Figure 6.1 119
6.4 CLP-based evaluator for the language defined in Figure 6.1 120
6.5 EBNF grammar defining the language the given parser accepts. 123
6.6 EBNF grammar defining the language students must define a parser for . 123
6.7 Equivalence classes of student submissions, based on which tests they failed. 126
6.8 Attempts to handle negative exponents, but incorrectly. 129
6.9 Implicitly assumes that the exponent will be non-negative. 129
6.10 Implicitly assumes that the exponent will be positive. 129

7.1 SimpleScala syntax. 136
7.2 Various definitions used in the typing rules. 138
7.3 Typing rules for SimpleScala, without pattern matching 141
7.4 Typing rules handling pattern matching in SimpleScala 142
7.5 CLP implementation of a typechecker for SimpleScala 143
7.6 Bounded version of the CLP code shown in Figure 7.5. 147
7.7 Version of the CLP code shown in Figure 7.6 149
7.8 A major improvement over the hole-filling strategy presented in Figure 7.7 150
7.9 Type derivation for a simple program using a cyclic "type" 154
7.10 The core logic of the `typeof` rule from Figure 7.6 after refactoring 166
7.11 The updated `typeof` rule from Figure 7.6 167

8.1 CLP snippet which is intended to handle the typing rules of a simple arithmetic language 176
8.2 Two procedures which perform an operation on an input list, yielding a new list 178
8.3 How the `map` operation can be implemented in Scala 179
8.4 Closest direct port of the Scala code in Figure 8.3 to CLP, using the built-in `call` procedure of CLP. 180
8.5 Abstract syntax of the portion of Typed-Prolog that interacts with Typed-Prolog's type system 184
8.6 Formalized definitions which are used throughout typechecking. 186
8.7 Typing rules over the syntax defined in Figure 8.5, using the typing domains defined in Figure 8.6 188
8.8 C code instrumented with higher-order capabilities 191
8.9 Defunctionalized version of the code shown in Figure 8.8 192
8.10 Typed-Prolog code using higher-order capabilities 193

8.11 Translation of the Typed-Prolog code in Figure 8.10 down to first-order CLP . . . 194
8.12 Module A, defined in a file named `modA.pl` . . . 198
8.13 Module B, defined in a file named `modB.pl` . . . 199
8.14 Result of compiling modules `modA` (in Figure 8.12) and `modB` (in Figure 8.13) together . . . 200

9.1 Simple generator for depth-bounded arithmetic expressions, consisting of the integers 0 and 1, with the operations + and − . . . 204
9.2 Basic, but nonetheless complete, CLP metainterpreter implementation . . . 211
9.3 Metainterpreter instrumented with depth-based bounding, based on the metainterpreter in Figure 9.2. . . . 215
9.4 Metainterpreter instrumented to only decrement the bound on calls to infinitely recursive procedures . . . 217
9.5 Metainterpreter which parameterizes the bounding operation, which generalizes the metainterpreter shown in Figure 9.4. . . . 219
9.6 Modified bounding operations useful for the metainterpreter in Figure 9.5. 221
9.7 Example query to the metainterpreter in Figure 9.5. . . . 222
9.8 Complete final metainterpreter . . . 226
9.9 Rewritten version of the code in Figure 9.7 . . . 229
9.10 Rewritten version of the code in Figure 9.9 . . . 230
9.11 Simplistic expression generator . . . 231
9.12 Refactored generation code from Figure 9.11 . . . 233

B.1 Example axiom and inference rule notation. . . . 263
B.2 Syntax for a variant of the simply-typed lambda calculus . . . 264
B.3 Typing rules for the syntax shown in Figure B.2. . . . 264

List of Tables

2.1 Generation rate of **sto** versus **clp**, in units of programs per second 40

3.1 Features of our data structures and properties 55
3.2 Description of bounds for all data structures under test 58
3.3 Performance data for small bounds 59
3.4 Performance data for medium bounds 59
3.5 Performance data for large bounds 60

4.1 Summary of reported issues and bugs, part I 86
4.2 Summary of reported issues and bugs, part II 87

5.1 Implemented test case generators, along with the SMT solvers they were used to test 106
5.2 The number and types of bugs found in each solver by each generator. . 110
5.3 The number and types of bugs found on a per-solver basis 110

6.1 Average code coverage metrics 130

Chapter 1

Introduction and Motivation

Software bugs are a costly plague. These have led to losses of millions of dollars [1], and even lives [2]. In order to prevent such catastrophic losses, we want to find these bugs as soon as possible, ideally **before** the damage has been done. Additionally, we want to find such bugs automatically, supplementing the sort of testing programmers already (should) perform. Such defines the area of *automated testing*, or *fuzzing*.[1]

Painting broad strokes, automated testing techniques can be divided into two categories: white-box and black-box. White-box techniques (e.g., [5, 6, 7, 8, 9, 10, 11]) require the source code of the system under test (SUT), whereas black-box techniques (e.g., [12, 13, 14, 15, 16, 17, 18]) do not require such source code. As one might expect, there are a number of research challenges in both categories, though the focus of this work is specifically on black-box fuzzing.

[1]The term "fuzzing" originally referred specifically to generating completely random inputs [3], as it was intended to be reminiscent of the word "noise" [4]. Over time, this word has been applied in a much wider context, making it largely synonymous with automated testing as a whole.

1.1 What is Black-Box Fuzzing?

The basic idea behind black-box fuzzing is to generate program inputs through some arbitrary process. As a simple example, one such process is to produce streams of random characters, as was done in Miller et al. [3]. Such inputs are then run on a SUT, and the SUT's response is used to determine whether or not the input triggered a bug. For example, if the SUT crashes, this likely indicates a bug. Similarly, if the SUT produces incorrect output (as determined by some sort of oracle), this likely indicates a bug.

While the approach of randomly generating streams of characters is technically sufficient for any problem, this approach is of little practical use in most domains. To understand why, consider the problem of testing a language implementation (e.g., a compiler or interpreter). Using random character streams as the generation model, it is expected that very few inputs are even syntactically valid. For example, consider the relatively tiny C snippet below:

```
1 int main(int argc, char** argv) {
2   return 0;
3 }
```

The above program, while syntactically valid and well-typed, hardly does anything interesting. Intuitively, only the buggiest of compilers is expected to have a problem with the above program, making it relatively useless as a test input. Nonetheless, the odds of producing the above program are astronomically low if the generator is based solely on random character generation. The vast majority of programs will fail to ever get beyond the parser, or perhaps even the tokenizer, making random character generation an unsuitable generation strategy for language implementations. For testing these sort of SUTs with structured inputs, it is necessary to use more sophisticated approaches. Such approaches are the topic of this thesis.

1.2 Why (Not) Black-Box Fuzzing?

Before diving into the details surrounding different black-box input generation approaches, we first look at why someone would want to use black-box fuzzing as opposed to white-box fuzzing.

Black-box fuzzing holds a number of advantages over white-box fuzzing. First and foremost, because black-box techniques do not need SUT source code, they can be readily applied to systems where the source code is unavailable or otherwise difficult to observe. This is generally true for proprietary systems without publicly-accessible source code. This is also true of SUTs which are written in multiple languages, as is common for Web applications. Overall, black-box techniques need only some mechanism to pass an input to a SUT, whereas white-box techniques need the capability to explore the SUT itself.

Beyond source code, black-box techniques generally scale better than white-box techniques [19], precisely because there is no coupling between the test case generator and the SUT. That is, because the test case generator is completely independent of the SUT, the size and complexity of the SUT is irrelevant to how difficult it is to generate SUT inputs. While SUT input generation may be a difficult task for black-box techniques, this task's difficulty is unrelated to the complexity of the SUT itself. In contrast, white-box techniques often struggle with complex SUTs, as white-box techniques closely depend on the SUT's structure. White-box techniques may require modification to work with complex SUTs (e.g., [11]), and even then certain SUT features are practically impossible to handle (e.g., hash functions [19, 11]).

To be fair to white-box techniques, black-box techniques suffer from their own problems. Arguably the largest problem is rooted in the decoupling of the testing technique from the SUT: black-box techniques rely on the law of large numbers (of inputs) to explore program behaviors, whereas white-box techniques methodically explore a SUT's

state space. This theoretically makes white-box techniques better-suited to finding edge cases which appear only under very specific inputs. While there is some work on augmenting black-box techniques with a more systematic search [20], such work is relatively still in its infancy.

1.3 Related Work

There is quite a bit of related work on generating inputs with some structure in a black-box fashion. We start this discussion with *stochastic grammars* [21], which serve as a more principled way of testing language implementations than the random character generation approach of Section 1.1.

1.3.1 Stochastic Grammars

The basic idea with a stochastic grammar is to perform a random walk over the SUT's input language grammar, producing AST nodes in the process. This grammar is assumed to be context-free. Such a process guarantees that output programs will be syntactically valid.

This stochastic grammar approach has been put to great use in testing dynamically-typed languages, particularly JavaScript. In particular, `jsfunfuzz` [14] uses this approach, and `jsfunfuzz` has found thousands of bugs in SpiderMonkey (Firefox's JavaScript engine) [22]. LangFuzz [13] is also based on this approach, and it similarly has found hundreds of bugs in SpiderMonkey [23]. LangFuzz also introduces additional capabilities to build new tests by *mutating* existing tests in random ways, which was intended to find bugs related to incomplete patches.

While stochastic grammars have been effective for testing dynamically-typed languages, they cannot be readily applied to statically-typed languages. Fundamentally,

stochastic grammars require the user to rephrase the language's type system as a grammar; that is, the user must define a grammar that emits not only syntactically valid terms, but also well-typed terms. While such definitions are feasible for simple type systems like that of C [21, 12], this quickly becomes impossible for more complex type systems. This is ultimately because typing rules generally require constraints which are more complex than what is strictly possible to express with grammars. In such cases, the user must either restrict themselves to a tiny subset of the language (as in St-Amour et al. [24]), or optimistically hope that most programs generated will happen to be well-typed (as in Daniel et al. [25]).

Stochastic grammars are similarly unequipped to reason about program *behavior*. For testing languages with undefined behavior (e.g., C and C++), this is a must, as programs which exploit undefined behavior are allowed to behave in any way possible. This makes such programs relatively useless for testing, as compilers may emit any possible output code (or not). As such, stochastic grammars alone are ill-suited to test languages with undefined behavior.

1.3.2 Typical Testing of Language Implementations Beyond Stochastic Grammars

The typical fuzzing approach here for adding additional structure is based on writing test case generators which are highly specific to the problem at hand. For example, Eclat [26] and JCrasher [27] are both able to generate well-typed Java programs by construction. This is significant, as the typing rules of Java are much more complex than those of C. However, careful inspection of the generation algorithms used reveals that this approach is applicable only to Java, and even then only to a relatively simplistic subset of Java. Brummayer et al. [17] similarly discusses a specialized algorithm for

generating well-typed formulas in the statically-typed SMT-LIB [28] language, though this algorithm cannot easily be extended to other languages. CSmith [12] is able to generate C programs which are guaranteed to be devoid of undefined behavior, but the approach used is highly complex and specific to C. Overall, while these approaches work for testing their respective intended SUTs, they are not intended to be general solutions.

1.3.3 Program Synthesis

Synthesis techniques (e.g., [29, 30, 31]) can produce well-typed programs with highly specific behaviors. However, to the best of my knowledge, synthesis techniques have never been applied to testing language implementations. This is because synthesis techniques are impractically slow for effective testing, as even relatively high-performance synthesis techniques only generate tens of programs per second. Additionally, synthesis techniques tend to operate over highly restricted languages, so they would hardly qualify as a general solution for the structured black-box test case generation problem.

1.3.4 Bounded-Exhaustive Test Case Generation

A more general solution is offered by work on *bounded-exhaustive test case generation*, wherein all the tests meeting user-defined constraints and fitting within provided bounds are generated. Depending on the particular system employed, these constraints can encode everything from basic syntactic validity to well-typed programs in non-trivial statically-typed languages.

A brief tour of work on bounded-exhaustive test case generation follows. TestEra [32] allows for test input constraints tobe specified in Alloy [33], which ultimately uses a SAT solver to find satisfying solutions for the provided constraints. These solutions can immediately be used as test inputs. While this approach is quite general, it is relatively

slow, as this entails many calls to a SAT solver.

In an attempt to improve the performance of such general test case generation, one of the TestEra authors later co-developed ASTGen [25]. In contrast to TestEra, ASTGen does not use a SAT solver, and constraints are instead encoded entirely in Java. This encoding is performed by painstakingly implementing iterator-like classes, which are used to build up valid test inputs by construction. The word "encoding" is used rather generously in this context; based on firsthand experience, it is difficult to encode anything more complex than a grammar using this approach. More complex encodings usually require a "generate-and-filter" approach, which entails generating arbitrary inputs and then filtering out invalid inputs post-hoc. As constraints become more complex, nearly everything generated is invalid, quickly cancelling out any performance gains as inputs are discarded.

A definite improvement over ASTGen is that of Korat [34], which again allows for constraints to be written in Java. However, unlike ASTGen, this encoding is performed through a predicate written in pure Java (i.e., a referentially-transparent method that returns `boolean`). This predicate takes a data structure as an input, and returns `true` if the data structure is valid, and `false` if it is invalid. With this predicate, Korat is able to effectively execute the Java code "backwards", that is, producing data structures for which the predicate returns `true`. Thanks to a clever generation algorithm, this is surprisingly fast. Overall, Korat is much faster than TestEra, and it is more expressive than ASTGen.

There were later attempts to improve the expessibility of ASTGen via UDITA [35], which composes ASTGen-style iterator-like classes with Korat-style predicates. While this work was sold somewhat as a way of offering the performance of ASTGen with the expressibility of Korat, experimentation in Dewey et al. [36] revealed that UDITA was uniformly slower than Korat, often by orders of magnitude. Later follow-up work

combining the advantages of TestEra and Korat exists [37], and this work has been shown to outperform both TestEra and Korat by even orders of magnitude. However, in order to see these performance advantages, it is necessary to implement the same constraints in **both** TestEra-style and Korat-style, increasing the difficulty of usage.

While the aforementioned work on bounded-exhaustive test generation seems applicable to relatively complex generation tasks like generating well-typed programs, we observe that this sort of task has not been well-explored. While both ASTGen [32, 25] and UDITA [35] have been applied to testing statically-typed languages (specifically Java), these applications have required post-hoc filtering out of ill-typed programs. As such, it is questionable as to whether or not these works are directly applicable to very complex generation tasks.

A radically different approach to bounded-exhaustive test generation is taken by Senni et al. [38, 39], which proposes the use of CLP [40, 41] for this purpose. This alone makes Senni et al. the most relevant work to that of my own. Senni et al. shows that CLP can be used to generate a variety of data structures, including red-black trees and heaps implemented with arrays. Moreover, Senni et al. shows that this generation is quite fast, often outperforming Korat [34] by orders of magnitude. However, Senni et al. fails to make any of the sort of key insights we make regarding the widespread applicability of CLP to the structured test case generation problem. Additionally, Senni et al. failed to push the expressibility bounds of CLP very far, whereas this thesis repeatedly and deliberately tries to push these bounds to the limit.

1.3.5 Fetscher et al.

The work of Fetscher et al. [42] is separated into its own category, given the unique nature of the work. This work builds a randomized test case generator on top of PLT-

Redex [43], which is used to effectively encode test input constraints in a Racket-based [44] domain-specific language. To date, this work has tackled the most complex generation task known outside of my own work. Specifically, Fetscher et al. was able to generate well-typed programs for a non-trivial, generic language (i.e., it had type variables) with higher-order functions. However, the rate of generation was impractically slow for efficient testing, with fewer than ten tests generated per minute.

1.4 Problem with Black-Box Fuzzing: Highly Structured Input Generation

While Section 1.3 shows that there is a great deal of related work in the area of black-box structured input generation, this is still not a solved problem. In particular, I observe the following issues with related work:

1. There is no single overarching theme. For example, other than the fact that both CSmith [12] and Fetscher et al. [42] can be used for randomized black-box testing of different SUTs, there appears to be no connection between the works. In particular, the various works which push the limits of stochastic grammars in Section 1.3.2 appear to all be unique in terms of approach.

2. When it comes to performance, there is little general understanding behind what makes things fast and what makes things slow. For example, while the work of ASTGen [25] and UDITA [35] argued that an iterator-based test generation style translated to high performance, this was thoroughly debunked in Dewey et al. [36] (see Chapter 3). Similarly, the results from Rosner et al. [37] show that even with the sort of repeated SAT solver calls needed in TestEra [32], TestEra can still nonetheless be faster than Korat [34], which is a more performance-minded

successor.

3. Most importantly, the bulk of the works cannot generate valid inputs with lots of structure, such as well-typed programs for a non-trivial type system. The few works that can perform such complex generation tasks are either entirely specialized for their SUT (e.g., CSmith [12]), or impractically slow (e.g., Fetscher et al. [42]).

The above observations collectively lead to my formal problem statement:

> ***Problem Statement:*** *In a generalized, black-box manner, how can we generate many instances of complex, structured test inputs for usage in software testing?*

1.5 Key Insights

With respect to the observations made in Section 1.4, there are several key insights which I have made. These insights serve as the foundation upon which I build my thesis. This section discusses these insights, along with their repercussions on any possible solution to the structured test input generation problem.

1.5.1 Insight #1: Logical Constraints can Describe Tests

The first insight is relatively minor: logical constraints can be used to represent test inputs. From this standpoint, different test inputs are different solutions to the same set of logical constraints. This is considered a minor insight because some tools already are based on using logical constraints to represent test inputs, such as TestEra [32].

As a thought experiment, we will extend this use of logical constraints to *all* test generation techniques, even those which do not use explicit constraints. While this is

admittedly a jump, constraints are present in other tools, even if they are not explicit. For example, we can view context-free grammars as a sort of restricted logic, where sentences in the language the grammar encodes correspond to true logical formulas. To see the rationality behind this observation, consider the example grammar in Figure 1.1. This grammar can be encoded in a straightforward manner using logical inference rules, as shown in Figure 1.2. While the logic in Figure 1.2 is relatively inexpressive, it is still nonetheless a logic. (As an aside, the inverse statement that logics define languages has been previously discussed in the literature, as in Kroening et al. [45], page 18.)

$$n \in \mathbb{N} \qquad b \in \mathit{Boolean}$$

$$\mathit{me} \in \mathit{NatExp} ::= n \mid \mathit{me}_1 + \mathit{me}_2$$
$$\mathit{be} \in \mathit{BoolExp} ::= b \mid \mathit{be}_1 \wedge \mathit{be}_2 \mid \mathit{me}_1 \leq \mathit{me}_2$$

Figure 1.1: Simplistic grammar handling expressions with natural numbers and booleans.

$$\frac{}{n : \textsf{NatExp}}\ (\textsc{Nat}) \qquad \frac{\mathit{me}_1 : \textsf{NatExp} \qquad \mathit{me}_2 : \textsf{NatExp}}{\mathit{me}_1 + \mathit{me}_2 : \textsf{NatExp}}\ (\textsc{Plus})$$

$$\frac{}{b : \textsf{BoolExp}}\ (\textsc{Bool}) \qquad \frac{\mathit{be}_1 : \textsf{BoolExp} \qquad \mathit{be}_2 : \textsf{BoolExp}}{\mathit{be}_1 \wedge \mathit{be}_2 : \textsf{BoolExp}}\ (\textsc{And})$$

$$\frac{\mathit{me}_1 : \textsf{NatExp} \qquad \mathit{me}_2 : \textsf{NatExp}}{\mathit{me}_1 \leq \mathit{me}_2 : \textsf{BoolExp}}\ (\textsc{Lte})$$

Figure 1.2: Encoding of the grammar in Figure 1.1 into equivalent logical inference rules.

The fundamentally logical nature of grammars is significant to the insight that logical constraints describe tests, considering that grammars serve as a common basis for testing tools which do not outwardly appear logical (see Section 1.3.1). With these tools, the general idea is to encode the language of the SUT as a grammar, and then the tool generates valid sentences of that grammar. From the logical perspective, this is equivalent

to encoding input constraints using a limited logic, and then using a constraint solver to generate solutions to the constraints. These solutions happen to encode sentences in the grammar at hand. While most of these tools also offer additional capabilities which cannot be formulated with context-free grammars alone, we can view such capabilities as additional kinds of encodable constraints in the tool's underlying logic. For example, CSmith [12] effectively adds in constraints describing the behavior of C programs.

1.5.2 Insight #2: Highly Specific Test Input Generation Demands Highly Expressive Logics

While grammars form a logic, this logic is quite inexpressive. For example, while grammars can accurrately encode the basic *structure* of a binary search tree (i.e., binary search trees contain integers, internal nodes, and leaves), they cannot accurrately encode the *contents* of a binary search tree (i.e., nodes appear in a specific order based on their values). As another example, grammars alone cannot accurrately encode the sort of constraints necessary to express well-typed inputs for most practical type systems, even relatively inexpressive type systems like that of Java. Without additional constraints, one must either conservatively give up on generating all possible well-typed programs (as is done in Eclat [26] and JCrasher [27]) , or alternatively resign oneself to generating some ill-typed programs (as was done with ASTGen [25] and UDITA [35]).

The subproblem of generating well-typed programs is of particular interest to this discussion. From a pragmatic level, this subproblem is well-motivated, as evidenced by the significant amount of related work in this area (e.g., [25, 35, 26, 27, 12]). From a theoretical level, this subproblem is particularly challenging. From the Curry-Howard correspondance [46], we know that type systems and logics are one in the same, where each type system is backed by its own logic. With this in mind, in order to generate

well-typed programs for an arbitrary language, theoretically we need an expressive logic with which to encode test inputs. Otherwise, the type system we want to encode may, in fact, be *more* expressive than the logic we try to encode it with, forcing us again to be imprecise with the encoding. With this in mind, in order to ensure our solution to this problem is general enough to handle well-typed program generation, we need a logic which is as expressive as possible.

1.5.3 Insight #3: Any Constraint Solver Employed Must Practically Generate Many Solutions

Beyond simply encoding logical constraints, some tool (hereafter referred to as a "solver") must exist to derive solutions to these constraints, as each solution ultimately forms a test input. In order to be practical, the solver must be capable of generating many solutions, as each solution forms only one test as part of a larger test suite. While the exact definition of "many" depends on both the sort of constraints in play and the SUT, in practice this tends to be on the order of tens of thousands to millions of tests. Considering that black-box test case generators are often run continuously (as is done with Z3 [47, 48] and Firefox [14, 22]), ideally there should be no upper limit on the number of tests a solver can produce. At the very least, if such an upper limit exists, it must be large enough that a sufficiently complex test suite can still be produced.

1.5.4 Insight #4: Any Constraint Solver Employed Must be Reasonably Fast

A solver must be able to generate these inputs reasonably fast. While this is somewhat related to the problem of generating many solutions, it is not quite the same problem. For example, while there may be no upper limit on the amount of solutions a solver

can produce, a solver which produces only one input per day is likely not very useful for generating test cases. Exactly what "reasonably fast" means depends again on the particular constraints and SUT; this usually means either that the solver finds lots of bugs in the SUT, or that the solver can generate tests at least as fast as they can be executed on the SUT. Based on my personal experiences, this usually means the solver must be capable of producing hundreds to thousands of tests per minute.

Acheiving sufficient performance is particularly difficult considering the need for maximum expressibility. Useful constraint satisfaction problems are often NP-Complete at best, and combining seemingly innocuous constraints together into a unified logic can quickly lead to undecidability. [45]

1.6 Potential Solutions

With the insights from Section 1.5 in hand, I argue that a solution to the structured black-box input generation problem **must** satisfy the following criteria:

- The solution must be based on encoding test inputs using logical constraints
- The logical constraint language must be expressive
- The solver for these constraints must be capable of generating many (100K+) inputs
- The solver for these constraints must be reasonably fast (100+ solutions per minute)

Ideally, I also want to reuse existing logics and constraint solvers as much as possible; the topic of this dissertation is black-box testing, not logic and constraint solver design. Fortunately, there are many existing logics and constraint solvers discussed in the literature. An assortment of these are discussed henceforth, along with their relative merits and problems.

1.6.1 SMT Solvers

SMT solvers [45] are incredibly popular for a wide variety of constraint satisfaction problems. For example, the paper for the Z3 SMT solver [47] has been cited over 1,000 times [49], and Z3 is but one of many SMT solvers. Over time, the state of the art in SMT solvers has improved dramatically, thanks in part to regular competitions [50, 51]. This improvement has led to dramatic increases in both performance and scalability, allowing modern SMT solvers to handle problems with thousands of variables and perhaps millions of constraints [45]. Additionally, solvers like Z3 [47] can regularly handle constraints over undecidable logics, making SMT solvers incredibly expressive.

For these reasons, SMT solvers may seem like viable solutions to the generalized structured black-box test generation problem. After all, their underlying logic for test inputs is remarkably expressive, and the solvers are quite fast. However, there is a major downside to SMT solvers for our purposes: they tend to be designed with the assumption that only **one** solution is needed, whereas we need hundreds of thousands for test generation. One can naively generate multiple solutions by adding *blocking clauses* to the original query [45], which effectively constrain the problem to generate a solution which differs from previously-generated solutions. One then queries the solver again with these blocking clauses, and iteratively continues this process of adding blocking clauses and solving until some set point. However, blocking clauses are problematic for two reasons:

1. As described, we must completely restart solving for each new set of blocking clauses. Considering that most solver work is expected to be in the initial query, in contrast to the blocking clauses, this is wasteful.

2. This approach fundamentally does not scale to many solutions. If we want k solutions, we will need to add in k sets of blocking clauses, where the size of each set

is bounded by the number of variables in the original problem [52]. With this in mind, the size of the blocking clauses will eventually exceed the size of the problem as k increases, and this may occur relatively early for variable-rich problems.

While the above problems can be at least somewhat overcome by modifying the solver itself (e.g., [53, 54]), such changes begin to blur the meaning of "SMT solver". Additionally, these changes cause their own problems. For example, while Z3 [47] has limited support for generating multiple solutions, using these capabilities can severely limit what the solver can practically reason about [48]. From personal experience, Z3 often becomes orders of magnitude slower if even two solutions are requested. As such, SMT solvers are overall inappropriate for solving the generalized structured black-box test generation problem.

1.6.2 ALLSAT Solvers

Unlike SMT solvers, ALLSAT solvers are designed from the ground-up to produce multiple solutions for a given query [52]. In this way, ALLSAT solvers are far superior to SMT solvers for solving the structured black-box test generation problem. However, ALLSAT solvers are nonetheless inappropriate for this problem.

The fundamental problem with ALLSAT solvers is that they are technologically primitive relative to SMT solvers. For one, the underlying logic of ALLSAT solvers is far more restrictive than that of SMT solvers. ALLSAT solvers are based on basic boolean satisfiability, whereas SMT solvers can handle more complex datatypes like mathematical integers and bitvectors [28]. This is likely not a fundamental limitation, as SMT solvers tend to be based around augmenting DPLL-based SAT solvers [45, 55, 56] with additional algorithms to handle more complex constraints [57]. However, either such algorithms have not been developed for the ALLSAT context, or they have yet to be

practically implemented.

A second problem related to the relatively primitive nature of ALLSAT solvers is that they are limited to too few solutions for our purposes. There is no competition for improving the state of the art in ALLSAT solvers (like that for SMT solvers [50]), though prerequisite steps have been performed recently [52]. Looking at the results in Toda et al. [52], modern ALLSAT solvers currently seem restricted to producing only a few thousand solutions total, with a generation rate of less than 200 solutions per minute. While the generation rate is fast enough for our purposes, the limitation to a few thousand solutions makes modern ALLSAT solvers impractical for testing large industrial applications, where we are expected to need hundreds of thousands to millions of inputs to sufficiently explore the code.

For these reasons, while ALLSAT solvers seem applicable to this problem in theory, they are still too primitive for my purposes. Until the state of the art in ALLSAT solvers improves, ALLSAT solvers are not a viable solution to the generalized structured black-box test case generation problem.

1.6.3 Answer Set Programming

As the name implies, answer set programming [58, 59] is geared towards generating multiple solutions for the same constraints. In this way, answer set programming is similar to ALLSAT solvers. However, the constraint language for answer set programming is far more expressive than that of ALLSAT solvers, and appears Prolog-like [60, 61] in construction. This makes answer set programming intuitively more attractive than ALLSAT solvers for the generalized black-box structured test input generation problem.

Unfortunately, much like with ALLSAT solvers, the state of the art in answer set programming is still primitive relative to SMT solvers. While there is an established

competition for improving answer set programming [62], it is younger than the equivalent competition for SMT solvers [50]. Looking closer at the answer set programming competition, it is currently biased towards incredibly difficult problems with fewer than 30 solutions. [62] Considering that test generation problems are usually simpler but with hundreds of thousands of inputs, we can extrapolate that this indicates that the current state of the art in answer set programming may not be appropriate.

We do not need to extrapolate far, however. Clasp [63, 64], a popular answer set programming implementation, also happens to have ALLSAT solver capabilities. Moreover, Clasp was evaluated in Toda et al. [52] alongside other ALLSAT solver implementations. All these implementations were deemed inappropriate for structured test case generation in Section 1.6.2. As such, assuming the answer set programming capabilities of Clasp behave similarly to its ALLSAT solver capabilities, then we can conclude that Clasp is overall inappropriate for this problem. Given that Clasp is a popular answer set programming language, it seems safe to assume that Clasp is indicative of the whole for answer set programming, so we can overall conclude that the state of the art in answer set programming is still too primitive for structured black-box test input generation.

1.6.4 Constraint Logic Programming

Much like SMT solvers, constraint logic programming (CLP) [40, 41] offers the user a highly expressive constraint language. However, unlike SMT solvers, CLP is well-suited to generating many solutions. Additionally, there is no fundamental limit to the number of solutions which can be generated, in stark contrast to the apparent limits seen with ALLSAT solvers (Section 1.6.2) and answer set programming (Section 1.6.3). CLP has been around longer than some of its competitors, and multiple high-performance implementations already exist (e.g., SICStus Prolog [65], GNU Prolog [66], SWI-Prolog [67],

and ECLiPSe [68]). For these reasons, CLP is the only solution discussed which satisfies all of the constraints of a necessary solution.

The one caveat of CLP is that the constraint language is so rich that it is Turing-complete. Because this can easily lead to situations where constraint satisfaction cannot proceed, CLP gives the programmer fine-grained control over how constraint search is performed. Specifically, CLP features a straightforward operational semantics which is simple enough to be described with a single inference rule (namely SLD-resolution [69]). This operational semantics makes the search behavior of CLP constraints not only predictable, but exploitable. CLP users can effectively "inject" domain knowledge into the code through optimizations. Such optimizations bear a striking resemblance to the approach of defining domain-specific constraint satisfaction algorithms (seen in Toda et al. [52]). However, these CLP optimizations show one major difference: these optimizations are in CLP code itself, as opposed to escaping CLP entirely and defining novel search strategies. This speaks to the applicability of CLP to a large variety of domains, as we do not need to define whole new search algorithms for each new domain.

This fine-grained control over search in CLP is both a blessing and a curse. This control allows CLP to be extended to wildly disparate testing domains and to problems which range the entire gamut in complexity. This generality comes at the cost of occasionally being difficult to use; the user may need to inject domain knowledge if things start getting slow, as opposed to simply picking a smarter implementation with a better search strategy as one might do with an SMT solver. That said, a smart enough engine might not exist (e.g., one may already be using the best SMT solver available), in which case one would almost assuredly be completely stuck without CLP.

1.7 Overarching Thesis

The insights in Section 1.5, along with the discussion of potential solutions in Section 1.6, lead me to my formal thesis statement:

> ***Thesis Statement:*** *We can represent structured test inputs as solutions to systems of logical constraints. In order to encode many kinds of test inputs and test a wide variety of systems, we need an expressive logic combined with a high-performance constraint solver capable of finding many solutions. I observe that CLP meets these criteria, and so CLP can ultimately solve the generalized structured black-box input generation problem.*

To be clear, solutions other than CLP are possible, though CLP is the only solution I am aware of which meets all the solution criteria defined in Section 1.6.

In order to demonstrate that CLP is a valid solution to the generalized structured black-box input generation problem, I must demonstrate that:

- The logic of CLP is expressive enough to encode a wide variety of test generation problems, including highly complex ones
- Some arbitrary CLP engine is fast enough to produce a sufficient amount of tests within a reasonable timeframe

To this end, I present a series of case studies wherein CLP was applied to test case generation in some particular domain. I have intentionally chosen a wide variety of domains, including those where test inputs have significantly complex structure. I submit this as evidence of the generality and expressibility of CLP for test case generation.

Moreover, where applicable, CLP was able to find real bugs in popular, industry-grade software. Additionally, at multiple points CLP was compared to direct competitors in

specific domains. Such comparison revealed that CLP is consistently able to generate test inputs orders of magnitude more quickly than that of the competition. I submit this as evidence that modern CLP engines are fast enough to solve the generalized structured black-box input generation problem, and overall have little difficulty producing the large numbers of inputs required for effective testing.

1.7.1 Organization of This Document

The rest of this document goes through each of the aforementioned case studies in detail. A brief description of each one of these case studies follows:

1. Chapter 2 shows how CLP can be applied to fuzzing dynamic languages, in particular JavaScript. This chapter also demonstrates that CLP is a strict generalization of the stochastic grammar approach described in Section 1.3.1. This chapter is based on work published in ASE'14 (Citation: [70]; DOI: `10.1145/2642937.2642963`; © 2014 ACM).

2. Chapter 3 shows how CLP can be applied to generating highly constrained data structures, well beyond anything ever previously attempted. Not only can CLP be applied to this problem, it was orders of magnitude faster than its competitors, namely Korat [34] and UDITA [35]. This chapter is based on work published in ICSE'15 (Citation: [36]; DOI: `10.1109/ICSE.2015.26`; © 2015 IEEE).

3. Chapter 4 shows how CLP can be applied to fuzzing the typechecker of the Rust programming language [71], which features a complex type system. This was the first work which ever attempted to efficiently generate tests for a type system of this level of complexity. During this process, 14 developer-confirmed bugs were found. This chapter is based on work published in ASE'15 (Citation: [72]; DOI: `10.1109/ASE.2015.65`; © 2015 IEEE).

4. Chapter 5 shows how CLP can be applied to fuzzing SMT solvers, via the generation of guaranteed satisfiable and unsatisfiable formulas. This process found 23 bugs, of which 22 have been fixed at this point. This chapter is based on work submitted (though not accepted) to ICSE'17.

5. Chapter 6 shows how CLP can be applied to fuzzing tokenizers and parsers, specifically in the context of finding bugs in student solutions to an educational problem. This process revealed CLP to be strictly better at finding bugs than a traditional manually-generated test suite, and exposed deficiencies in the student-provided assignment specification. This chapter is based on work published in ITiCSE'17 (Citation: [73]; DOI: `10.1145/3059009.3059051`; © 2017 ACM).

6. Chapter 7 shows how to apply CLP to generating well-typed programs in a non-trivial language, specifically in an educational context. This chapter is written somewhat in a tutorial style, and offers lots of code samples for a complex generation problem. This chapter is unpublished.

The last two chapters (specifically Chapters 8 and 9) discuss certain limitations of CLP-based test generation, along with how these limitations are overcome. While this work has not been published, it has assisted me during multiple testing projects over the years. Although these latter chapters do not directly contribute to my thesis statement, they are so closely connected to the case studies described that I felt it necessary to include them. Additionally, these chapters highlight certain technical and engineering details which are relevant to using CLP for test case generation. As such, these chapters are absolutely relevant to using CLP for testing, even if they do not directly defend the thesis statement.

The rest of this thesis assumes the reader has at least a passing familiarity with CLP; readers who are not familiar with CLP should consult Appendix A. Mathematical

formalisms are presented in multiple areas of this thesis; details behind the notation used are available in Appendix B.

Pronouns Used in This Document

Both "I" and "we" are used in this document. Uses of "I" refer to thoughts and work which I consider entirely my own. However, much of this work was produced through collaboration and discussion with others, particularly with the people mentioned in the acknowledgements section. While I am the main author of all the work in this document, I do not claim total overship over such collaborative portions, and so I use "we" for such cases. Occasionally I use a *royal* "we" as well, referring to things in a more general sense.

Chapter 2

Case Study: Generating Interesting JavaScript Programs

2.1 Introduction

This chapter features a case study wherein CLP is applied to the generation of JavaScript programs with predictable runtime behaviors. The purpose of this case study is to demonstrate that CLP can be used to not only implement stochastic grammars (discussed in Chapter 1, Section 1.3.1), but to implement program generators *strictly more expressive* than that possible with stochastic grammars alone. The sort of CLP-based generators defined in this chapter can perform simple but effective semantic reasoning about JavaScript programs, which is well-beyond anything possible with a syntax-oriented approach like that of stochastic grammars.

In regards to the thesis, this chapter demonstrates that CLP is expressive, fast, and capable of generating many programs. Specifically, CLP is significantly more expressive than the otherwise popular stochastic grammars, to the point where CLP can accurrately reason about the semantic properties of generated programs. Despite the added express-

ibility, CLP is still nonetheless able to generate thousands to hundreds of thousands of programs **per second**.

As an aside, this chapter is based on work which we published in ASE'14 (Citation: [70]; DOI: `10.1145/2642937.2642963`; © 2014 ACM).

2.2 CLP for Program Generation

This section discusses how to use CLP for program generation. We will begin with a syntactic description of a simple arithmetic expression language and express progressively more interesting properties for expression generation. Section 2.3 will then discuss applying these ideas to program generation specifically for JavaScript.

2.2.1 Syntactic Expressions

We use the arithmetic expression language shown in Figure 2.1 for the examples throughout this section. We can describe this grammar in CLP, as shown in Figure 2.2.

$$i \in \mathbb{Z}$$
$$e \in \mathit{ArithExp} ::= i \mid e_1 + e_2$$

Figure 2.1: Basic arithmetic expression language, consisting of integers and addition over expressions.

As shown in Figure 2.2, two clauses are used in the `exp` procedure, one for each of the alternative productions of *ArithExp* in Figure 2.1. The first clause (on lines 1-3) checks that the input (`I`) is an integer between `INT_MIN` and `INT_MAX`, where `INT_MIN` and `INT_MAX` are assumed to be defined elsewhere with the obvious meanings. The second clause (on lines 4-6) checks for the case of $e_1 + e_2$, which is assumed to be represented as a structure named `add` with two parameters representing e_1 and e_2, respectively. Because

```
1  exp(I) :-
2    INT_MIN #=< I,
3    I #=< INT_MAX.
4  exp(add(E1, E2)) :-
5    exp(E1),
6    exp(E2).
```

Figure 2.2: CLP implementation of the grammar shown in Figure 2.1

e_1 (`E1`) and e_2 (`E2`) should themselves be expressions, `exp` is recursively used on these parameters to ensure that they are, in fact, expressions (on lines 5-6).

We can use the CLP definition in Figure 2.2 in two distinct ways: to *recognize* valid expressions and to *generate* valid expressions. In the recognition case, we pass a potential expression as an argument to the `exp` procedure and it will return either `true` if it is a valid expression or `false` otherwise. The more interesting case for fuzzing is generation: if we want to generate valid expressions instead, we can use a query like the following:

```
1  ?- exp(E),
2     write(E),
3     fail.
```

With the above query, the CLP engine will first attempt to find a value for the logical variable `E` that will make the `exp` procedure true on line 1. Once such a value is found, the engine will write that value to output on line 2, then fail on line 3. Failure will cause the engine to backtrack and attempt to find a *different* value for `E`; this process will continue indefinitely and generate an infinite stream of valid expressions.

One caveat is that the resulting expressions will not be concrete; instead, they will contain *symbolic variables* (standing for unknown integers) that are subject to a set of *constraints* derived from the clauses. For example, one of the generated expressions would be `add(X, add(Y, Z))` where `X`, `Y`, and `Z` are symbolic variables standing for unknown integers. The CLP engine remembers the bounding constraints on each one of these

variables (i.e., $\forall_{a \in \{\texttt{X},\texttt{Y},\texttt{Z}\}}$ `INT_MIN` $\leq a \wedge a \leq$ `INT_MAX`). To get a concrete expression, we need to then ask the CLP engine to *label* the symbolic variables, that is, find concrete values that satisfy the constraints. The engine guarantees that satisfying values must exist.

2.2.2 Bounding Size

CLP engines use unbounded depth-first search by default. This strategy ultimately controls the order in which expressions are generated. For an infinite stream of expressions (as would be produced by the generator in Figure 2.2), the search strategy controls exactly *which* expressions are generated within a finite amount of time. While the search strategy is effectively "baked-in" the engine, we can still exercise simple control over it. For example, consider the following query:

```
1 ?- call_with_depth_limit(exp(E), 5, CurrDepth),
2    CurrDepth \== depth_limit_exceeded,
3    write(E),
4    fail.
```

The above query uses `call_with_depth_limit` on line 1, which is a built-in procedure in the SWI-PL [67] CLP engine. This procedure runs a given query (`exp(E)` above) with a bound on the recursion depth (`5` above). If the query ever exceeds this bound, it will unify a given variable (`CurrDepth`) with the atom `depth_limit_exceeded`. This can then be used as a flag to check whether or not the query exceeded the bound, as is checked in line 2 above. The code on line 2 will cause failure to occur if the bound was exceeded (i.e., this checks to see that `CurrDepth` does *not* hold the atom `depth_limit_exceeded`). If the depth is not exceeded, `CurrDepth` will instead hold an integer indicating how deep the recursion went. With all this in mind, the above query effectively bounds the depth of the expressions generated.

This can further be used to implement an iterative deepening search strategy, if so

desired. Using iterative deepening, we can generate *minimal* test cases which find bugs. This is in stark contrast to the bulk of related work, which relies on random generation of large programs (e.g., [14, 13, 12, 21, 74]). While such large programs find bugs, they tend to contain lots of unrelated code, necessitating downstream test case reduction techniques (e.g., [75, 76, 77, 78, 79, 80, 81, 82]).

While the bounding approach described here based on `call_with_depth_limit` works for stochastic grammars, it is tied to SWI-PL [67]. Additionally, it does not scale well to larger problems, and it is difficult to bound based on elements other than recursion depth without significant code changes. Chapter 9, Section 9.2.2 describes this problem in detail. Chapter 9 also offers a general solution in Section 9.4.1.

2.2.3 Stochastic Grammars

CLP subsumes the stochastic grammar technique for program generation. This means that we can specify stochastic grammars using CLP, and do so quite easily. An example showing such stochastic behavior is shown in Figure 2.3, which was derived from the CLP code in Figure 2.2. The key to the stochastic behavior in Figure 2.3 is the use of the `maybe` procedure, which triggers random failure with a given probability. This leads to a somewhat random distribution of programs being produced, as is expected from stochastic grammars.

It should be noted that while the `maybe`-based technique described is generally sufficient for stochastic grammars, this tends not to scale well with more complex generators. The reasons why, along with solutions to this problem, are discussed in Chapter 9, specifically Sections 9.2.1 and 9.4.3.

```
1  exp(I) :-
2    maybe(0.4),
3    INT_MIN #=< I,
4    I #=< INT_MAX.
5  exp(add(E1, E2)) :-
6    exp(E1),
7    exp(E2).
```

Figure 2.3: CLP code from Figure 2.2 augmented with stochastic capabilities, thanks to the built-in `maybe` procedure in the SWI-PL [67] CLP engine. The use of `maybe` above triggers random failure with 40% probability, allowing for random exploration similar to that of a stochastic grammar.

2.2.4 Arithmetic Overflow

One of the most powerful abilities of CLP is symbolic arithmetic reasoning, which enables the user to specify numeric constraints as part of a predicate. These constraints are handled by an integrated constraint solver as part of the CLP engine. For example, we can specify that we only want to generate expressions that contain at least one arithmetic overflow, shown in Figure 2.4. Stepping through this code, the `exp` procedure (derived from Figure 2.2) has been instrumented with a second integer parameter holding the result of the given expression, along with a third boolean parameter indicating whether or not overflow has occurred. Lines 5-7 are almost identical to lines 1-3 in Figure 2.2, except that now `exp` indicates that overflow did not occur in line 5. Lines 8-16 are more interesting, because these dicate how arithmetic addition is performed. Line 11 performs the addition, though in a symbolic way with `#=`. If the result of this operation is greater than `INT_MAX`, then overflow (`Over`) is set to `true` in line 12. Similarly, if the result of this operation is less than `INT_MIN`, then overflow is set in line 13. If, however, the result is between `INT_MIN` (line 14) and `INT_MAX` (line 15), we do not immediately set that overflow occurred. Instead, this will return `true` (indicating overflow occurred) if either of the subexpressions overflowed, using the `boolOr` helper prodedure. Only if the result

was between `INT_MIN` and `INT_MAX`, **and** if none of the subexpressions cause overflow to occur, will the result of the addition indicate overflow.

```
 1  boolOr(true, _, true).
 2  boolOr(_, true, true).
 3  boolOr(false, false, false).
 4
 5  exp(I, I, false) :-
 6    INT_MIN #=< I,
 7    I #=< INT_MAX.
 8  exp(add(E1, E2), N, Over) :-
 9    exp(E1, N1, Over1),
10    exp(E2, N2, Over2),
11    N #= N1 + N2,
12    ((N #> INT_MAX, Over = true);
13     (N #< INT_MIN, Over = true);
14     (N #>= INT_MIN,
15      N #=< INT_MAX,
16       boolOr(Over1, Over2, Over))).
```

Figure 2.4: CLP code snippet which can generate programs exhibiting arithmetic overflow. The second parameter of the `exp` procedure indicates the integer result of evaluating the arithmetic expression. The third parameter of the `exp` procedure dictates if overflow occurs in the expression generated (the first parameter to `exp`), with `true` indicating overflow occurred and `false` indicating otherwise. This is based on the CLP code in Figure 2.2.

An example query using the code in Figure 2.4 to generate expressions that contain at least one overflow is shown below:

```
1  ?- exp(E, _, true),
2     write(E),
3     fail.
```

The above query will generate some expression `E` which is required to overflow somewhere, as indicated by the third parameter to `exp` being `true` in line 1. In this case, we do not care about the actual result of the expression, indicated by the use of underscore (`_`) for the second parameter to `exp` in line 1. It will then write out the expression on line 2 and

trigger backtracking for another expression on line 3, much like the query shown for the original CLP code in Figure 2.2.

2.3 Generating JavaScript

This section discusses how to fuzz dynamic languages with CLP, using JavaScript as a representative example. JavaScript is an imperative, dynamically-typed language with objects, prototype-based inheritance, higher-order functions, and exceptions. JavaScript is designed to be as resilient as possible and makes liberal use of implicit conversions and other idiosyncratic behaviors. Object properties can be dynamically inserted and deleted, and when performing a property access the specific property being accessed can be computed at runtime. There are several mechanisms available for runtime reflection. Object inheritance is handled via delegation: when accessing a property that is not present in a given object *obj*, the property lookup algorithm determines whether *obj* has some other object *proto* as its prototype; if so then the lookup is recursively propagated to *proto*. We omit exact details of the CLP predicates that we use for JavaScript, but they are available in this chapter's supplementary materials.[1]

2.3.1 Stochastic Grammar

It is fairly simple to construct a CLP predicate describing syntactically valid JavaScript programs; it is a straightforward extension of the method we used for the example arithmetic expression language in Section 2.2. Just as we did there, we can add probabilities to the clauses of that predicate in order to convert it to a stochastic grammar. This yields a program generator much in the spirit of `jsfunfuzz` [14].

[1] `https://www.cs.ucsb.edu/~benh/research/downloads/ase14_supplementary.zip`

However, while a purely stochastic grammar works for finding bugs in dynamic languages, we observe that most of the programs it generates are not particularly useful for testing. The problem is that no matter what language features are present in a generated program, it is likely that there will be a runtime type error during execution before most of those features are ever reached. For example, any attempt to access a property of the `null` or `undefined` JavaScript values will raise an exception and terminate the program's execution (unless the exception is caught and handled, which again is unlikely in a randomly generated program).

2.3.2 Absence of Runtime Errors

We can use CLP to specify JavaScript programs that avoid runtime errors. The idea is to use CLP to encode a type system and compose the syntactic JavaScript predicate with a predicate that only matches on well-typed programs. One nice property of this technique is that we can *incrementally* make the type system more powerful, starting with a simple system and adding precision as needed. We can also choose to make the type system either sound (restricting the programs that can be generated, but ensuring that all such programs will avoid runtime errors) or unsound (allowing more programs to be generated, but potentially allowing some kinds of runtime errors). Thus we have two axes of freedom to work with, allowing us a great deal of flexibility to trade off between the effort required to write program generators and the kinds of programs that will be generated.

For example, we could employ a simple, unsound type system as a first effort to help avoid some runtime errors. Figure 2.5 shows a fragment of a simple type system that prevents programs from directly dereferencing the `null` JavaScript value; however, it still allows dereferencing a variable which itself may have the value `null`. We can encode this

type system in CLP, as shown in Figure 2.6.

$$\frac{}{\vdash \texttt{null} : \textsf{nil}} \qquad \frac{}{\vdash x : \textsf{unk}} \qquad \frac{\vdash e_1 : \textsf{unk}}{\vdash e_1.e_2 : \textsf{unk}}$$

Figure 2.5: A fragment of an unsound type system for JavaScript to filter out programs that attempt to directly access a property of the `null` value.

```
1  typeof(null, nil).
2  typeof(var(_), unk).
3  typeof(access(E1, _), unk) :-
4    type(E1, unk).
```

Figure 2.6: CLP code implementing the type system fragment shown in Figure 2.5.

To step through the code in Figure 2.6, the predicate `typeof` takes two parameters: a JavaScript expression and the type of that expression, respectively. Line 1 states that the expression `null` has type **nil**, representing an expression which evaluates to `null`. Line 2 states that any variable (where the underscore is a placeholder for the variable name) has type **unk**, where **unk** represents some unknown, possibly `null` value. Line 3 states that if we attempt to access an object held in expression `E1`, and the type of `E1` is unknown (**unk**) on line 4, then the type of the access overall is **unk**. This procedure would disallow programs such as "`null.foo`" (which we want to eliminate, as this just triggers an error), but still allow programs such as "`var x = null; x.foo`" (which is still undesirable, though less likely of a program given its added complexity). If there are too many runtime errors in the resulting programs we can extend this type system to track the types of variables more closely, either soundly or unsoundly depending on how precise we want to be and how much effort we want to put into it.

2.3.3 Arithmetic Overflow

Arithmetic overflow is an interesting property for testing JavaScript implementations. JavaScript numeric values are technically floating point, though for performance reasons most JavaScript engines try to represent numeric values as integers wherever possible. Therefore, at runtime the engine must detect all overflows and automatically change the numeric representation from integers to floating point values. We can extend the arithmetic overflow predicate described in Section 2.2 to generate valid JavaScript numeric expressions that are guaranteed to contain at least one overflow, in order to test whether the engine performs this optimization correctly.

2.3.4 Prototype-based Inheritance

Object are the fundamental data structure in JavaScript (even functions and arrays are special kinds of objects), and so testing object inheritance is key. To do so, we must generate programs that both *construct* and *use* non-trivial prototype chains. Stochastic grammars are unlikely to generate such programs by themselves. Using CLP, we can enforce that generated programs contain the following items in sequence, in the proper scope, potentially with unrelated code in-between:

- A declaration for some function *foo*.
- A statement *foo*`.prototype.`*fld* `=` *exp*, where *fld* is some string and *exp* is a valid JavaScript expression, seeding *foo*'s prototype with the property *fld*.
- A statement `var` *x* `=` `new` *foo*`()`, constructing a new object whose prototype is the same as *foo*'s.
- A statement *x*`.`*fld*, triggering prototype lookup.

A high-level view of the CLP code implementing this behavior is shown in Figure 2.7. In Figure 2.7, the `inSequence` procedure ensures that its arguments happen in sequence, however they are not necessarily consecutive, i.e., an arbitrary number of other expressions may occur in-between them. This sequence specifies a function declaration with a given name `Name`, followed by an optional setting of that function's `prototype` field, followed by an optional setting of a field of that function's prototype, followed by the rest of the AST.

```
1 usesPrototypeBasedInheritance(S) :-
2   inSequence([declareFunctionWithName(Name),
3               giveFunctionPrototype(Name),
4               addToFunctionPrototype(Name),
5               =(Rest)], S),
6   astContains(Rest, useNewWithFunctionWithPrototype(Name)).
```

Figure 2.7: CLP code implementing a generator of JavaScript expressions which use prototype-based inheritance. While the definitions of `inSequence`, `declareFunctionWithName`, `giveFunctionPrototype`, `addToFunctionPrototype`, `astContains`, and `useNewWithFunctionWithPrototype` have not been provided, these can be implemented to do what their names suggest. This code was pulled directly from the implementation, with slight modifications related to naming and the removal of extraneous capabilities which distract from the main discussion.

We can extend this specification in various ways to make for even more interesting tests. For example, we could (among many other possible options):

- Require a minimum depth for the prototype chain
- Require the use of implicit conversion to convert non-objects to objects and then perform prototype lookup on the resulting objects
- Construct multiple interleaved prototype chains
- Implement some or all of the above in conjunction with each other

2.3.5 With + Closures

JavaScript has some very obscure and idiosyncratic behaviors that implementations must get correct. One example is the combination of the `with` expression with closures. The `with` expression changes the current scope, affecting variable lookup. Closures should preserve the current scope, but closures and `with` have complex interactions that make the proper behavior unclear and difficult to get correct. A particularly tricky case is when a closure is created inside of a `with` expression, and then subsequently called outside of that `with` expression.

With this in mind, we can specify a CLP predicate that generates JavaScript programs with the following requirements:

- The program contains a `with` expression that itself contains a closure definition.
- The function definition contains multiple free variables that are defined in various scopes with respect to the `with` scope and the scope outside of the `with`.
- The closure value formed from that function is passed outside of the `with` and subsequently called.

A high-level view of this predicate is shown in Figure 2.8.

```
1 withAndClosureEdgeCase(S) :-
2   inSequence([defineClosureWithNameInWith(Name),
3               callClosureWithName(Name)], S).
```

Figure 2.8: CLP code implementing a generator of JavaScript expressions which use `with` combined with closure creation. While the definitions of `inSequence`, `defineClosureWithNameInWith`, and `callClosureWithName` have not been provided, these can be implemented to do what their names suggest. This code was pulled directly from the implementation, with slight changes to procedure names to improve clarity.

In Figure 2.8, `defineClosureWithNameInWith` ensures that the generated program contains a `with` expression containing a closure and `callClosureWithName` ensures that

the closure is called outside of the `with`. The variable `Name` is used to track the name of the variable that holds the closure, ultimately allowing `callClosureWithName` to call the proper closure created by `defineClosureWithNameInWith`. This is an example of how CLP allows the tester to concentrate on a specific set of language features and a specific interaction between those language features.

2.4 Evaluation

This section evaluates the effectiveness of CLP-based program generation versus a stochastic grammar approach. We first explain our experimental methodology, then we present and discuss our results for JavaScript.

2.4.1 Methodology

We compare two different approaches to program generation for language fuzzing:

1. **sto**: The stochastic grammar approach
2. **clp**: The CLP-based approach

The comparison is a bit misleading because CLP can implement stochastic grammars; in fact, all of the approaches are implemented in SWI-Prolog [67], a publically-available CLP implementation. The comparison showcases the additional expressiveness and efficiency of CLP over purely stochastic grammars.

At the time this case study was performed, publically-available implementations of the existing JavaScript fuzzers `jsfunfuzz` [14] and LangFuzz [13] did not exist. While we requested such implementations from Mozilla for comparison to our technique, our requests were ignored. As such, we did not compare against these tools.

The entire fuzzing infrastructure, including the code implementing the **sto** and **clp** approaches described above, is available in the supplementary materials for this chapter.[2] These experiments were run on a system with 12 Intel Xeon@1.9 Ghz cores and 32 GB memory. All experiments are single-threaded.

The metric that we use to measure program generation effectiveness is *generation rate*, in terms of programs per second. We measure three distinct kinds of generation rate: **total**, **unique**, and **interesting**, with the following meanings:

- **total** rate: how quickly each approach can generate programs, without regard to what kinds of programs are being generated.
- **unique** rate: how quickly each approach can generate unique programs, ignoring duplicate ASTs. For this metric, unless otherwise stated, variable names and the values of number, string, and boolean literals are irrelevant. For example, the programs `x + 6` and `y + 7` would be considered identical.
- **interesting** rate: how quickly each approach can generate programs that match some tester-given criteria for being interesting. In our case, *interesting* means that the program is accepted by one of our predicates discussed in Section 2.3.

To compute the generation rate for each approach, we use the given approach to generate programs for five minutes and then divide the resulting number of programs by 300 seconds to get units of programs per second. We do this separately for each kind of generation rate (**total**, **unique**, and **interesting**). We report separate **total**, **unique**, and **interesting** numbers for the **sto** approach; the **clp** approach only generates unique, interesting programs and so we only report the **interesting** number for that approach.

To be clear, our generation metric is not intended to focus on how fast programs can be generated by the various techniques, but rather to reveal how well the techniques can

[2] `https://www.cs.ucsb.edu/~benh/research/downloads/ase14_supplementary.zip`

be tuned to generate interesting programs for various definitions of "interesting", and how easily they can be targeted for different interesting properties.

Details of the sto Approach

For JavaScript, we create a predicate `exp` that describes syntactically valid programs. We then add probabilities as discussed in Section 2.2 to create a stochastic grammar, carefully tuned to favor the generation of unique programs. Querying this predicate using:

```
1  ?- exp(E),
2     write(E),
3     fail.
```

... will yield a stream of randomly generated, syntactically valid programs. This method is equivalent to the current state of the art for stochastic grammar-based approaches (see Chapter 1, Section 1.3.1).

Details of the clp Approach

The CLP approach uses the predicates specifying interesting programs directly in order to generate satisfying programs; thus the generated programs are guaranteed to be both unique and interesting. We set the search strategy used by the CLP approach for all predicates to bounded depth-first search. Again, while our experiments treat **sto** and **clp** as distinct approaches, we should be clear that in reality **clp** subsumes **sto** and that these approaches can all be easily combined together in various ways.

2.4.2 JavaScript Program Generation

We evaluate four predicates for generating JavaScript programs:

1. `js-err`: generates programs with few runtime errors, relative to a traditional syntactic fuzzer. This uses the general technique described in Section 2.3.2, though with a more extensive type system. Specifically, the type system attempts to ensure that (1) expressions that are used for property access or as the subject of a `delete` are neither `null` nor `undefined`; and (2) expressions that are called as functions, either directly or via `new`, are actually function values.

2. `js-overflow`: generates programs that contain at least one arithmetic overflow, utilizing the technique described in Section 2.3.3. We consider two ASTs which contain an overflow at the exact same position to be the same.

3. `js-inher`: generates programs that construct and use prototype-based inheritance chains, utilizing the technique described in Section 2.3.4.

4. `js-withclo`: generates programs that construct closures inside of `with` expressions and then call them outside of those `with` expressions. This utilizes the technique described in Section 2.3.5.

Table 2.1 gives the generation rates for **sto** compared to **clp**.

Predicate	**total**	**unique**	**interesting**	**clp**	$\frac{\textbf{clp}}{\textbf{interesting}}$
`js-err`	53,335	28,614	24,210	56,658	2.3
`js-overflow`	49,635	22,631	0.303	1,229	4,056
`js-inher`	20,677	11,195	0	304,890	∞
`js-withclo`	31,458	17,132	0.057	302,472	5,306,526

Table 2.1: Generation rate of **sto** versus **clp**, in units of programs per second. We report **total**, **unique**, and **interesting** separately for **sto**; they are all the same number for **clp**. The last column is the ratio between the **clp** and **sto interesting** program generation rates (higher is better for **clp**).

We can group the predicates based on their behaviors, with `js-err` in one class and `js-overflow`, `js-inher`, and `js-withclo` in the other class. For `js-err`, fully 85% of

the unique programs are interesting. While **clp** generates slightly fewer total programs than **sto**, it generates 2.3× as many interesting programs during the same period of time. This is the best that **sto** does in comparison to **clp**, and the two major reasons are that (1) for this predicate we are using an unsound type system, thus it is easier for the stochastic grammar to generate programs matching the predicate; and (2) for this predicate we bound the search space for both **sto** and **clp** to programs whose abstract syntax trees are at most height seven. Both of these facts favor **sto** in our experiments; by making the type system more sound or by increasing the bound on program size, **clp** would do progressively better than **sto**.

The remaining three predicates have a very different story. The **interesting** generation rate for **sto** is 0 or very close to 0, while the generation rate for **clp** for two of the predicates is 10–15× higher than **sto**'s **total** generation rate. The reason that **clp** is so much faster than **sto** in this case is because **sto** is randomly searching the space, i.e., each time it finds a program it restarts the search from scratch. In contrast, **clp** is using bounded depth-first search, i.e., when it finds a program it searches the nearby space to find other programs as well. This behavior of **sto** is characteristic of stochastic grammars, not specific to our implementation. The **clp** generation rate for `js-overflow` is much lower than the **clp** generation rate for the other predicates because `js-overflow` heavily exercises SWI-PL's [67] arithmetic constraint solver, which tends to be expensive.

2.5 Conclusions

This chapter has demonstrated that CLP is strictly more expressive than stochastic grammars, and can even reason about the expected runtime behaviors of generated inputs. However, this generation is nonetheless fast and capable of generating thousands of programs per second. Moreover, this was done for a realistic kind of SUT, namely

JavaScript engines. With this in mind, the benefits of CLP are quite practical in nature. All these points help to defend my thesis that CLP is an effective solution to the structured black-box input generation problem.

Of potential interest to the reader is that the paper this chapter is based on (Dewey et al. 2014 [70]) mentioned two future directions:

1. The application of CLP to fuzzing statically-typed languages
2. The integration of SMT solvers [45] into CLP

Both these directions were explored in later case studies; Chapters 4 and 7 both discuss the fuzzing of statically-typed languages bearing complex type systems, and Chapter 5 integrated SMT solver support into CLP for the purposes of better fuzzing SMT solvers.

Chapter 3

Case Study: Generating Complex Data Structures

3.1 Introduction

In this chapter, a case study is provided wherein CLP is applied to the generation of complex data structures. While there is a significant amount of prior work on generating data structures using test case generation tools (namely everything described in Chapter 1, Section 1.3.4), this chapter observes that all of the data structures previously generated have been relatively simple. In order to push the state of the art forward and ultimately test the expressiveness of CLP, this chapter looks at data structures never before generated, with a focus on those which are particularly challenging to generate.

To push the complexity level even further, for each data structure considered, we also look at a particular subset of the data structures which are of interest to testing. These subsets are defined by particular properties the data structures have, and they tend to involve the actual *operations* on the data structures themselves. For example, instead of merely generating red-black trees, we also generate red-black trees which will

internally rebalance upon the insertion of a particular value. This requires us to reason not only about valid red-black trees, but also the operations on valid red-black trees, namely insertion and rebalancing. Merely encoding this can be a challenge, let alone generating data structures that satisfy this encoding. As such, this serves as an excellent case study to test the expressibility limits of CLP.

In addition to pushing the expressibility limits of CLP, this chapter seeks to determine exactly how fast CLP is, relative to the competitors of Korat [34] and UDITA [35]. Since both Korat and UDITA are already designed to generate data structures, we can compare CLP to them directly, as long as we write equivalent generation code in Korat and UDITA. Because we could unintentionally write a poor Korat or UDITA-based generator, we intentionally used as many existing generators as possible for these tools, where the existing generators were written directly by Korat and UDITA authors. This allows for a comparison between generators optimized for Korat/UDITA against generators optimized for CLP.

This case study has overall revealed that CLP is capable of generating such complex data structures, which is encouraging considering that we chose data structures which are more complex than ever before generated. Additionally, the generation rate of CLP has shown itself to often be **several orders of magnitude greater** than that of Korat or UDITA, and CLP was uniformly at least several times faster than the competition.

As an aside, this chapter is based on work we published in ICSE'15 (Citation: [36]; DOI: `10.1109/ICSE.2015.26`;). While that work discussed the relative usability of the different test case genertaion tools employed, along with significant discussion around the terms "imperative" and "declarative", this chapter intentionally avoids such discussion. Such discussion distracts from the main purpose of this thesis, namely to demonstrate that CLP is an effective solution to the structured black-box test case generation problem. In particular, that work focused on exactly *how* one can encode

data structures, as opposed to what is merely *possible* to encode. While these two ideas are related (i.e., something may be possible to encode but so unrealistically difficult that it may as well be impossible), none of these data structures showed any extremes this severe.

This chapter begins with a discussion of exactly how CLP relates to the sort of existing work on data structure generation, first introduced in Chapter 1, Section 1.3.4.

3.2 Example

This section shows an example where CLP is used to generate sorted linked lists, shown in Figure 3.1. While sorted linked lists are not a particularly complex data structure, this still illustrates the basic concepts.

```
1 sorted([]).
2 sorted([_]).
3 sorted([A,B|Rest]) :-
4   A #=< B,
5   sorted([B|Rest]).
6
7 ?- sorted(L),
8    write(L),
9    fail.
```

Figure 3.1: CLP-based generator of sorted linked lists.

In Figure 3.1, there are three clauses followed by a query. The first clause (on line 1) states that an empty list is a sorted list. The second clause (on line 2) states that a single-element list is a sorted list. The third clause (on line 3) states that a multi-element list is sorted if the first two elements in are ascending order and the rest of the list is sorted. The query (starting on line 7) will indefinitely write out sorted lists according to the `sorted` procedure.

3.3 CLP Compared to Other Data Structure Generators

In this section, a number of features which are useful for data structure generation are discussed, along with how these features work in CLP and other data structure generation tools. I argue that CLP is constructed from the ground-up with these features in mind, whereas these features tend to have ad-hoc implementations in other tools, if they are present at all.

3.3.1 Feature: Nondeterministic Search

From a high level, all prior work can be seen as techniques to nondeterministically generate data structures of interest in a given space. In practice, they all employ various forms of backtracking algorithms. In TestEra [32], ultimately the nondeterministic generation is done by SAT solvers, which use backtracking algorithms for search [45]. However, given that traditional SAT/SMT solvers are not well-suited to generating many solutions (as described in Chapter 1, Section 1.6.1), this can be slow. In Korat [34] a backtracking algorithm is added on top of the JVM and used to search the space of all structures to find ones that match a predicate defined by the user. UDITA [35] actually modifies the semantics of Java using Java PathFinder [83], which uses backtracking to make Java execution nondeterministic. These techniques all build nondeterministic search into the infrastructure, hiding it from the user. ASTGen [25], in contrast, forces the user to explicitly encode the nondeterministic search into the specification of the data structure being generated, via careful composition of imperative iterator-like [84] objects.

Nondeterministic execution is a core feature of CLP semantics, and has been discussed in the literature since very early on [60]. As such, there have been literally decades of

work on making this feature efficient in CLP engines.

3.3.2 Feature: Search Strategy Control

All prior work has employed bounded-exhaustive generation, i.e., defining a finite space and generating all structures within that space. While bounded-exhaustive search has merit [85, 86], there are other search strategies that can be useful. Random search, iterative deepening, and various hybrid approaches have been used in the past to good effect for other types of automated generation (e.g., [12, 13, 14, 87]). As such, restricting search to a single strategy is overly limiting.

ASTGen and UDITA allow for some coarse-grained control over the order of data structure generation, but CLP can easily exceed this low bar. This capability of CLP was previously discussed in Chapter 2, Section 2.2; it is further discussed in Chapter 9.

3.3.3 Feature: Equality Constraint Propagation

Senni et al. [38] observe that UDITA's [35] lazy data structure instantiation optimization "can be seen as a particular CP (constraint propagation) solution strategy". We observe that this UDITA optimization in fact behaves just like the logical variables available in typical CLP engines, which allow for the propagation of *equality constraints*. Semantically, logical variables start in a special *uninstantiated* state, wherein the variable has no specific value. Uninstantiated variables can be aliased with each other, essentially putting variables into the same equivalence class. Logical variables can later become *instantiated* with particular values, and all aliased variables will automatically have that same value. To better illustrate this phenomenon, consider the following code: `X = Y, Y = 1`. This code aliases the logical variables `X` and `Y` with the expression `X = Y`, then sets both of them to the value `1` with the expression `Y = 1`. This behavior bears striking

similarity to the lazy instantiation optimization in UDITA, which: (1) only instantiates variables when operations specific to a given data structure are performed on them, and (2) allows for uninstantiated variables to be aliased. In this way, UDITA is attempting to emulate the logical variables already available in CLP engines, though in an ad-hoc manner.

3.3.4 Feature: Disequality Constraint Propagation

In Korat [34], blind search is avoided by observing what sort of sub-structure caused a data structure to be rejected by the user-defined predicate. This information is retained in a way that prevents further data structures with identical sub-structure from being generated. We observe that, in effect, this strategy allows Korat to propagate *disequality constraints*, which prevent the generation of invalid sub-structures. In UDITA [35], certain optimizations related to isomorphism-breaking are implemented in a manner which is similar to disequality constraint propagation. This observation is made directly by the authors in demonstrating the correctness of the details of their generation algorithm. While disequality constraints are somewhat non-standard in CLP languages, they are still compatible with CLP [88].

3.3.5 Arithmetic Constraint Propagation

The hallmark of CLP engines is the ability to reason about symbolic arithmetic via high-performance arithmetic constraint solvers. While all of the prior work allows for generation of data structures with arithmetic invariants, with the exception of Senni et al. [38] this capability is handled via a generate-and-filter approach. That is, instead of asking a constraint solver to deliver numbers which satisfy some given arithmetic constaints, one must instead try all numbers in a range and filter out those which did

not satisfy applicable arithmetic invariants. Not only is this inefficient, it forces the data structure generator to reason about data structure shape and contents simultaneously, which can be problematic. For example, consider the problem of generating sorted lists of length 0 to n. In general, there are only $n+1$ unique list shapes in this space, though a potentially infinite number of list structures when content is taken into account. If the tester desires to test only structures with particular shapes, without regard to contents, the space is quite small. However, the need for contents can blow up the search space in a completely uninteresting direction. With CLP, it is possible to reason about shape and contents independently and to request only a single satisfying solution for a list of any given length. With the prior techniques, it would be necessary to tweak various bounds in an ad-hoc manner just to get a single solution, and this sort of tweaking does not scale to arbitrary data structures.

3.4 Data Structures and Properties

This section describes the seven data structures on which we evaluate the competing data structure generation techniques. In addition to the baseline data structure definitions, we also describe for each data structure an additional property that targets an interesting part of the space of such structures; these additional properties are intended to further stress the test case generation tools being evaluated. Three of the structures have been evaluated in prior work, and are included here for comparison: sorted linked lists, red-black trees, and heaps. Four of the structures have never been evaluated for generation before this work: image grammars, skip lists, splay trees, and B-trees. The additional properties for all of the structures, including the three structures seen in prior work, are novel to this work.

Here we informally describe the structures and properties. The code used to gen-

erate these data structures for CLP, Korat, and UDITA is available in this chapter's supplementary materials.[1]

3.4.1 Data Structure: Sorted Linked Lists

A sorted linked list is a linked list whose nodes are ordered according to their contents, in this case integers. Both UDITA [35] and Senni et al. [38] generated these data structures.

Additional Property: We target lists where each integer element is separated by at most a value of k. For example, a valid list for $k = 3$ would be `[0, 2, 5, 5]`.

3.4.2 Data Structure: Red-Black Trees

A red-black tree is a type of balanced binary search tree that is commonly used as an efficient representation for sets and maps. Korat [34], UDITA [35], and Senni et al. [38] all showed that they could generate red-black trees with varying degrees of success.

Additional Property: We target red-black trees such that inserting a given element is guaranteed to cause rebalancing. Intuitively, this means that given an element value, we generate red-black trees such that if the given element is inserted into the tree it will cause a rebalance to occur. This property requires us to encode the insertion and rebalancing operations in our code that describes acceptable red-black trees.

3.4.3 Data Structure: Heaps

A heap is a type of balanced binary tree that is commonly used to represent priority queues efficiently. While heaps are usually described as trees, they are often backed

[1] `https://www.cs.ucsb.edu/~benh/research/downloads/icse15.zip`

by arrays, and so we choose an array-based representation. As with red-black trees, Korat [34], UDITA [35], and Senni et al. [38] all generated these structures.

Additional Property: We target heaps which require exactly $\log_2 n$ operations on `dequeue`, where n is the number of nodes in the heap. Such data structures show worst-case behavior, and are interesting not only for testing but for benchmarking. This property requires encoding the `dequeue` operation in the code describing acceptable heaps.

3.4.4 Data Structure: ANI Images

Chapter 2 showed that CLP is well-suited to generating data structures describable by context-free grammars, namely programs. In order to increase the complexity, here we choose a data structure that obeys a context-*sensitive* grammar, namely ANI images [89]. Bugs in parsers for this grammar have been historically costly [90]. We observe that the ANI grammar is not well-documented, and that there are several edge cases where it is unclear if a parser should accept or reject a given image. For our standard definition of ANI images we avoid these edge cases.

Additional Property: We target specifically those edge cases that we avoid in the standard definition. In other words, we target ANI images that are guaranteed to contain at least one edge case. Specifically, these edge cases are:

- A `Rate` subsection named "`LIST`", which introduces a parsing ambiguity with another image component with the same name.
- An `InfoList` subsection of size 2, which should not be possible with valid data.
- A `title` or `author` field holding a non-printable character.
- An image containing no icons (indicated with an icon length of 0), which are core components of an image.

- A `jifRate` of 0, which corresponds to an animation that would move infinitely fast.

3.4.5 Data Structure: Skip Lists

A skip list [91] is a special DAG-like representation of a linked list that allows for multiple elements in a list to be traversed in a single operation. The consequence of this on peformance is that inserting an element into a sorted linked list can be performed in $\mathcal{O}(\log\ n)$, unlike the typical $\mathcal{O}(n)$. Of special interest is that these data structures rely on probabilistic features and thus do not have deterministic shapes.

Additional Property: We target skip lists where fewer than $k\%$ of the elements have the maximum height. The observation this property is based on is that the smaller this percentage becomes, the less likely the data structure is in practice (due to the probabilistic features of the skip list algorithm), and thus we are more likely to generate what can be considered an edge case. Ideally we would like a very small percentage, though this percentage also influences the number of elements in the tree. To keep list sizes manageable, we use $k = 25$.

3.4.6 Data Structure: Splay Trees

Splay trees [92] are a type of binary search tree whcih automatically reconfigure themselves upon access. This reconfiguration intuitively makes elements which are frequently accessed cheaper to access. Central to this reconfiguration is a `splay` operation that moves a given element to the root of the tree via a series of modifications.

Additional Property: Due to the `splay` operation, the shape of a splay tree can vary widely between different operations which call `splay`. For testing, we are interested in particularly dramatic changes to the tree's shape. Specifically, we want to generate trees for which the following two properties hold in conjunction, where n is the total number

of nodes in the tree:

- The tree contains at least one node at depth greater than $\lceil 1.5 \times \log_2(n) \rceil$.
- If a `splay` operation is performed on any single node in the tree, all nodes in the tree would have depth $\leq \lceil 1.5 \times \log_2(n) \rceil$.

The aforementioned properties define splay trees which can become more balanced via some particular use of `splay`. Generating such splay trees would be useful for testing any optimization scheme based on this observation.

3.4.7 Data Structure: B-Trees

B-trees [93, 94] are a complex tree-based data structure which are used heavily in databases and filesystems. Given the fact that these are so popular at base system levels, it is useful to be able to generate these automatically for testing purposes.

Additional Property: We take a similar approach as with red-black trees, generating trees which would experience node-splitting given some particular value to insert.

3.4.8 Relevant Data Structure Features

From the above data structures and properties, we have derived a set of features that impact data structure generation and which any automated data structure generation technique must be able to handle. These properties, along with why they are potentially concerning, are described below.

Arithmetic

Structures and properties that require reasoning about arithmetic present problems for techniques that do not employ arithmetic constraint solvers. This includes all tech-

niques described other than CLP. While these past techniques *can* generate structures that require arithmetic reasoning, the lack of a constraint solver means that generation is extremely inefficient.

Probabilistic

Structures and properties that require probabilistic features may be difficult to represent. Directly encoding probabilities, e.g., using a random number generator, is not possible for all techniques. For example, TestEra [32] cannot handle probability at all, and Korat [34] requires that the predicate describing a data structure must be deterministic. For such frameworks, a tester is forced to either encode directly what low-probability data structures look like (which may not be obvious), or overapproximate by generating all possible data structures.

Graph-like

We define *graph-like* data structures to mean those structures which cannot be represented directly with trees or lists, which we refer to as being *tree-like*. Even so-called "trees" can be graph-like when they include parent and/or sibling pointers. Graph-like data structures are somewhat challenging to represent in CLP, as these require indirection to implement. That is, because CLP lacks assignment, we cannot simply update pointers and make a potentially cyclic data structure.

Loops and Assignment

For the data structures with operational variants, these operations are usually described via loops and assignment. However, CLP lacks these features, which requires the generator writer to rethink exactly what these operations are trying to accomplish.

Worst-case scenario, the writer might need to emulate these operations, as through a *store-passing* transformation (somewhat akin to the `State` monad in Haskell [95]).

Features in Summary

The information regarding these different features and their importance to data structure generation techniques is summarized in Table 3.1.

Property / Operation	**Seen In**	**A?**	**P?**	**G?**	**L?**
Numeric Ordering	sorted linked lists, red-black trees, heaps, skip lists, splay trees, B-trees	✓	✗	✗	✗
Explicit Tree	red-black trees, grammars, splay trees, B-trees	✗	✗	✗	✗
Rebalancing	red-black trees, heaps*, splay trees, B-trees	✓	✗	✓*	✓
Probabilistic Shape	skip lists	✗	✓	✗	✗

Table 3.1: Features of our data structures and properties and whether they bear attributes relevant to how automated data structure generation is performed.
A = Uses **A**rithmetic
P = **P**robabalistic
G = **G**raph-like
L = Uses **L**oops and assignment
*Heaps are not graph-like, though the rest of the listed data structures are.

3.5 Evaluation

In this section, we evaluate and compare Korat, UDITA, and CLP for performance and scalability. With respect to the overarching thesis, the goal of this evaluation is to determine if CLP can express all the data structures and variants described in Section 3.4, and to see how rapidly CLP can generate these structures.

3.5.1 Experimental Methodology

We have specified basic versions of the seven data structures described in Section 3.4 and also advanced versions containing the additional properties, in each of Korat, UDITA, and CLP. To be concise, we uniformly refer to these 14 versions as "data structures". The basic data structure is referred to as "basic", and the version of the data structure with the additional property is referred to as "special".

To measure performance we record the time each technique takes to generate all structures within a given set of bounding values. While evaluations in prior work report bounds as a single uniform value n, we observe that this does not reflect reality for even the simplest of data structures. That is, for all the data structures involved, there are multiple distinct bounding values that must be specified. Therefore, we report all of the bounding values used for each data structure. A description of these bounding values is provided in Table 3.2. Henceforth we will refer to these bounding values via comma-separated lists of integers, where the integer's position reflects which bound is being referred to in Table 3.2 and the integer value is the actual bound. For example, with basic sorted lists the bounds "2, 3" would mean a maximum of two nodes, and a maximum element value of three. Additionally, we set $k = 3$ for special sorted lists (see Section 3.4) and we ensure that we insert an element distinct from the tree contents for special red-black trees and special B-trees. To measure scalability, we break the performance results into three separate groupings based on small, medium, and large bounding values. For CLP, we chose GNU Prolog [96, 66] as our engine due to its public availability and high performance.

Where possible we used publicly available code for our specifications; if Section 3.4 mentions that an implementation of a data structure on Korat or UDITA was available, then we used that code. Such code reuse helps to lessen the threat that we unintentionally

wrote a slow Korat or UDITA representation, as this reused code was written by the authors of Korat and UDITA themselves.

3.5.2 Performance Results and Discussion

Performance data for Korat, UDITA, and CLP generators for small, medium, and large bounds are shown in Tables 3.3, 3.4, and 3.5, respectively. There are several key points to make with this data, which are brought out in order of increasing bounds.

Poor UDITA Performance

As shown in Table 3.3, even with small bounds UDITA often takes orders of magnitude more time than Korat. According to the prose found in the literature [25, 35], UDITA is faster than ASTGen which itself is implied to be much faster than Korat. We initially thought that our setup must be in some way malformed, but upon further examination this observation turns out to be consistent with results reported separately for Korat [34] and UDITA [35]. For example, while Korat and UDITA both evaluate on heap arrays of length 8 in the two works cited above, UDITA is a full order of magnitude slower than the same result in Korat, despite the fact that UDITA was introduced nearly 8 years after Korat. UDITA was never directly evaluated against Korat, though it was evaluated against its predecessor ASTGen, which is implied to be faster than Korat (though this is never evaluated empirically). Overall, this leads us to conclude that UDITA performs poorly in general.

Korat on B-Trees

Given the overall performance results in Tables 3.3, 3.4, and 3.5, an unexpected datapoint in Table 3.3 is that Korat takes orders of magnitude more time than UDITA with

Data Structure	Bound 1	Bound 2	Bound 3	Bound 4	Bound 5
Sorted Lists	Max # of Elements	Max Element Value	—	—	—
Red-Black Trees	Max # of Internal Nodes	Max Node Value	—	—	—
Heaps	Max # of Internal Nodes	Max Node Value	—	—	—
Images	Max # of Icons	Max Title Length	Max Author Length	Max `cSteps`	Max `jifRate`
Skip Lists	Max Height	Max # Elements	Max Element Value	—	—
Splay Trees	Max # of Internal Nodes	Max Element Value	—	—	—
B-Trees	Max # of Nodes	Max Element Value	—	—	—

Table 3.2: Description of bounds for all data structures under test. In all cases, the minimum element value is 0.

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	6, 6	0.266	898	0.001
Special Sorted Lists	6, 6	0.226	898	0.001
Basic Red-Black Trees	10, 10	45.8	322	0.3
Special Red-Black Trees	10, 10	24.7	327	0.168
Basic Heaps	8, 8	5.6	335	0.96
Special Heaps	8, 8	4.8	347	0.036
Basic Images	2, 1, 1, 1, 2	12.9	1027	2.8
Special Images	1, 1, 1, 1, 1	41	1611	7
Basic Skip Lists	4, 3, 3	67.4	1461	0.001
Special Skip Lists	4, 3, 3	58.4	1467	0.001
Basic Splay Trees	5, 5	1.86	66.8	0.001
Special Splay Trees	4, 4	0.386	81.8	0.001
Basic B-Trees	2, 2	140	1.05	0.001
Special B-Trees	2, 2	3.64	1	0.001

Table 3.3: Performance data for small bounds, in seconds.

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	12, 13	62.5	—	1.61
Special Sorted Lists	12, 13	51.5	—	0.38
Basic Red-Black Trees	12, 12	1218	—	1.26
Special Red-Black Trees	12, 12	709	—	0.804
Basic Heaps	9, 9	55.1	—	6.32
Special Heaps	9, 9	45.6	—	1.18
Basic Images	2, 1, 1, 2, 2	908	—	37
Special Images	2, 1, 1, 2, 2	—	—	279
Basic Skip Lists	4, 4, 4	—	—	0.001
Special Skip Lists	4, 4, 4	—	—	0.001
Basic Splay Trees	6, 6	96.7	—	0.016
Special Splay Trees	6, 6	361	—	0.004
Basic B-Trees	4, 4	—	—	0.001
Special B-Trees	4, 4	—	—	0.001

Table 3.4: Performance data for medium bounds. "—" signifies timeout after 1800 seconds (30 minutes).

Data Structure	Bounds	Korat	UDITA	CLP
Basic Sorted Lists	17, 17	—	—	569
Special Sorted Lists	18, 18	—	—	734
Basic Red-Black Trees	18, 18	—	—	189
Special Red-Black Trees	20, 20	—	—	560
Basic Heaps	11, 11	—	—	873
Special Heaps	12, 12	—	—	937
Basic Images	20, 1, 1, 2, 2	—	—	691
Special Images	10, 1, 1, 2, 2	—	—	1792
Basic Skip Lists	9, 9, 9	—	—	1136
Special Skip Lists	9, 9, 9	—	—	810
Basic Splay Trees	11, 11	—	—	487
Special Splay Trees	12, 12	—	—	380
Basic B-Trees	10, 20	—	—	67.5
Special B-Trees	20, 20	—	—	102

Table 3.5: Performance data for large bounds. "—" signifies timeout after 1800 seconds (30 minutes).

basic B-trees, though its performance improves by orders of magnitude when considering special B-trees. This performance is due to the fact that Korat must generate arrays all at once—Korat cannot piece arrays together incrementally, in contrast to UDITA. Our B-tree specification relies heavily on arrays and constraints on arrays, and so Korat ends up generating many unsatisfiable arrays in relation to UDITA. Korat on special B-trees is much faster because there is only one satisfying structure in the space. In other words, the vast majority of the structures in this space are invalid—since Korat learns from negative information, it very quickly learns the fact that the space is nearly entirely unsatisfiable, and is able to skip most of that space without having to explore it.

Excellent CLP Performance

For all data structures and for all bounds, CLP outperforms Korat and UDITA, usually by orders of magnitude. This is best shown in Table 3.5, which features large bounds for these data structures. As shown, with large bounds, Korat and UDITA

timeout on all data structures, whereas CLP does not timeout on anything.

There are several reasons why CLP offers such good performance. With CLP we have more direct control over the search strategy. For example, consider B-trees, which have an invariant that all leaves must be on the same level. With Korat, the best we can do is assert that this is true and hope that the Korat engine learns the pattern. With CLP, we directly constrain the generation so that all leaves are on the same level by construction, and so that they are only put at positions where they could legally be with respect to the number of nodes in the tree. This ends up cutting down dramatically on the amount of unsatisfiable search space.

A second major reason for CLP's performance is the presence of an arithmetic constraint solver. As an example, consider the performance results for the additional property on red-black trees in Table 3.4. While the property intuitively requires a generate-and-filter approach, we can actually do better with CLP. CLP allows us to impose symbolic arithmetic constraints on the data structure which force rebalancing to occur with respect to a given key, even without knowing the concrete values in the tree. This means that once the constraints are imposed, it is a simple matter of enumerating all integers which satisfy the symbolic constraints, which is a relatively low-cost operation. In contrast, with both Korat and UDITA, we are forced to take a generate-and-filter approach.

A third reason for the performance gains is that with CLP, we are implicitly utilizing decades of research into making nondeterministic search and arithmetic constraint solvers fast. We get these benefits "for free" just using the existing GNU Prolog engine.

3.5.3 Threats to Validity

In cases where existing data structure generation code was not already available from the authors of Korat [34] and UDITA [35], we had to write our own generators. The

performance of these tools can be sensitive to the coding style used, so it is conceivable that our generators for these data structures could be further optimized.

3.6 Conclusions

This chapter has shown that CLP can be used to express data structures more complex than ever before generated. Most important to the overall thesis is the fact that CLP was typically **orders of magnitude faster** at generating data structures than its direct competitors, namely Korat and UDITA. As far as performance is concerned, this serves as strong evidence that CLP is an effective solution to the structured black-box test case generation problem.

Chapter 4

Case Study: Type-Based Fuzzing of the Rust Compiler

4.1 Introduction

This chapter discusses a case study wherein guaranteed well-typed and ill-typed Rust programs were generated with the help of CLP. Such programs were then used to test the typechecker in the Rust [71] compiler, revealing 14 developer-confirmed bugs in the process. While finding these bugs was certainly beneficial, merely generating Rust programs with known typedness (that is, programs we know to be well-typed or ill-typed by construction) is a feat in and of itself. Rust's type system includes features like parametric polymorphism and generics; only Fetscher et al. [42] (described in Chapter 1, Section 1.3.5) can even handle these features properly, and even then the results are impractically slow for effective testing. Additionally, Rust's type system features affine types for proving memory safety at compile time without garbage collection; this is the first work which has ever even attempted to generate programs containing affine types with known typedness. Overall, generating Rust programs with known typedness pushes

the expressibility bounds of CLP, as an inexpressive logic would be unable to accurrately represent Rust programs with known types.

The results of this case study show that CLP was able to represent well-typed and ill-typed Rust programs with relative ease. In fact, informally, most of the difficulty in this case study was merely in determining *what* Rust considers to be well-typed and ill-typed, independent of CLP. Not only was CLP able to encode this information, the resulting test case generator was quite fast, with tens of millions of programs generated per hour. This was significantly faster than we could run the tests, even in a highly parallel setting with a heavily-optimized test harness. Additionally, CLP was able to generate hundreds of millions of tests overall, and no slowdown to the aforementioned generation rate was ever observed.

For the above reasons, this case study serves as strong evidence for my thesis, namely that CLP is a proper solution to the structured black-box test case generation problem. The generation of Rust programs with known typedness is of a complexity level beyond anything ever attempted before, but CLP was nonetheless able to express this. Moreover, the resulting generator was so fast that it outstripped the actual testing. The fact that 14 developer-confirmed bugs were found in the process shows that CLP is not only useful, but that the problem I am trying to solve is firmly rooted in reality: CLP has helped find bugs in popular, industrial-grade software with hundreds of millions of related downloads [97].

As an aside, this chapter is based on work we published in ASE'15 (Citation: [72]; DOI: `10.1109/ASE.2015.65`; © 2015 IEEE).

4.2 Generating Well-Typed Programs

We first discuss exactly how CLP can be used to construct well-typed programs. We use as an example the problem of generating well-typed programs in System F [98],

the polymorphically-typed lambda calculus. The relatively simple System F is used for exposition purposes only; CLP is capable of handling much more complex type systems, as demonstrated by the application of these ideas to Rust in Section 4.4. No existing fuzzers can handle something even as simple as System F because of its higher-order functions and parametric polymorphism.

While CLP is a better solution for generating well-typed programs for language fuzzing than any current method, it is not a perfect solution. We conclude this section by describing some of the pitfalls of CLP with respect to well-typed program generation.

4.2.1 CLP Example: System F

The best way to explain how to use CLP for well-typed program generation is by example. Here we demonstrate how to use CLP to generate well-typed programs in a variant System F, the polymorphically-typed lambda calculus. We make one change to System F, namely the addition of built-in support for integers, which occasionally makes generation more convenient (see Appendix C for details). The key points to observe are that:

- The CLP specification closely mirrors the formal type system definition.
- We use only the standard features of CLP languages, which means that we can use off-the-shelf CLP implementations to generate programs.

Figure 4.1 describes the syntax of the variant of System F we will use. The possible types are integers (**integer**), type variables (α), function types ($\tau_1 \to \tau_2$), or polymorphic types ($\forall\alpha.\tau$). An expression is either an integer, a variable, a function abstraction, a function application, a type abstraction, or a type application. Type abstractions parameterize an expression by a type in the same way that function abstractions parameterize

$$i \in \mathbb{Z}$$
$$\tau \in \mathit{Type} ::= \textbf{integer} \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$$
$$e \in \mathit{Exp} ::= i \mid x \mid \lambda x{:}\tau\,.\,e \mid e_1\ e_2 \mid \Lambda\alpha\,.\,e \mid e\ \tau$$

Figure 4.1: Syntax for System F, where i is an integer, α is a type variable, $\forall\alpha.\tau$ is a polymorphic type, x is a program variable, $\Lambda\alpha\,.\,e$ is a type abstraction that creates an expression of polymorphic type, and $e\ \tau$ instantiates a polymorphic expression to a specific type τ.

an expression by a value. Type application specializes an expression (the body of a type abstraction) to a given type in the same way that function application specializes an expression (the body of a function abstraction) to a given value.

$$\frac{}{\Gamma \vdash i : \textbf{integer}}\ (\textsc{int}) \qquad \frac{x \in \texttt{keys}(\Gamma) \quad \tau = \Gamma(x)}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash (\lambda x{:}\tau_1\,.\,e) : \tau_1 \to \tau_2}\ (\textsc{abs})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}\ (\textsc{app})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\Lambda\alpha\,.\,e) : \forall\alpha.\tau}\ (\textsc{tabs}) \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau_2 \quad \tau_3 = \texttt{subst}(\alpha, \tau_1, \tau_2)}{\Gamma \vdash (e\ \tau_1) : \tau_3}\ (\textsc{tapp})$$

Figure 4.2: Typing rules for System F, where Γ is a type environment mapping variables to types and $\texttt{subst}(\alpha, \tau_1, \tau_2)$ substitutes the type τ_1 for free occcurances of α in type τ_2, yielding a new type.

Figure 4.2 describes the typing rules for System F. The first four rules (INT, VAR, ABS, and APP) are exactly the same as in the simply-typed lambda calculus. The last two rules handle polymorphism: TABS introduces a polymorphic type and TAPP eliminates a polymorphic type.

Figure 4.3 shows a translation of the formal typing rules from Figure 4.2 into CLP. Figure 4.3 contains five clauses, one for each typing rule in Figure 4.2. The last line shows a query which will endlessly generate well-typed programs. The `typeof` procedure represents a type judgement: `typeof(Gamma, E, T)` stands for $\Gamma \vdash e : \tau$.

```
 1 typeof(_, int(_), integer).
 2 typeof(Gamma, variable(X), T) :-
 3   lookupMap(Gamma, X, T).
 4 typeof(Gamma, lam(X, T1, E), arrow(T1, T2)) :-
 5   addMap(X, T1, Gamma, NewGamma),
 6   typeof(NewGamma, E, T2).
 7 typeof(Gamma, app(E1, E2), T2) :-
 8   typeof(Gamma, E1, arrow(T1, T2)),
 9   typeof(Gamma, E2, T1).
10 typeof(Gamma, tlam(A, E), poly(A, T)) :-
11   typeof(Gamma, E, T).
12 typeof(Gamma, tapp(E, T1), T3) :-
13   typeof(Gamma, E, poly(A, T2)),
14   substitute(A, T1, T2, T3).
15
16 ?- typeof([], E, T), writeln(E), fail.
```

Figure 4.3: CLP specification of System F, where `lookupMap`, `addMap`, and `substitute` are helper procedures with the obvious functionality whose definitions are not shown here. The final query will output an infinite stream of well-typed System F programs.

The type environment Γ is represented using a list associating variables with their types; we use the helper procedure `lookup(Gamma, X, T)` to determine what type `T` is associated with variable `X` in type environment `Gamma`, and the helper procedure `addMap(X, T, Gamma, NewGamma)` to compute a new type environment `NewGamma` copied from the original type environment `Gamma` but associating variable `X` with type `T`. We also use the helper predicate `substitute(A, T1, T2, T3)` to compute a new type `T3` derived from type `T2` but with all free instances of type variable `A` replaced with type `T1`, i.e., $\texttt{T3} = \texttt{subst}(\texttt{A}, \texttt{T1}, \texttt{T2})$.

While the `lookupMap`, `addMap`, and `substitute` operations are not particularly lengthy, their implementation comes with technical challenges. Such challenges are not specific to generating well-typed Rust programs, and so they are beyond the scope of this chapter. Chapter 7 discusses these sort of challenges along with the implementations of these

operations, particularly in Section 7.3. A full implementation of the generator in Figure 4.3, along with the `lookupMap`, `addMap`, and `substitute` procedures, can be seen in Appendix C.

4.2.2 Pitfalls of CLP

While there are substantial advantages to using CLP for well-typed program generation, it does have certain problems that make it an imperfect solution. We identify and describe the two biggest problems we have encountered when using CLP for this purpose.

Fundamental Performance Issues

Typing rules generally assume that they are operating over complete programs and are attempting to make a judgement whether that program is well-typed, i.e., they are operating as *acceptors* rather than *generators*. Ideally, when implemented in CLP any acceptor is also a generator by default—given a predicate p that describes terms with some desired property, $p(t)$ operates as an acceptor for a concrete term t while $\exists x.p(x)$ operates as a generator that will bind x to some satisfying concrete term. However, naively translating typing rules into CLP can lead to performance issues. For example, while the ordering of clauses in a conjunction is irrelevant from a strictly logical standpoint, in practice it is significant. A poor ordering can lead to asymptotically worse performance [99], or even nontermination [100]. We have also found that it may be necessary to place bounds on non-expression components like types in order to ensure termination. Chapter 7 discusses these sort of problems more, particularly in Sections 7.3 and 7.4.

Lack of Constructive Negation

CLP lacks the ability to constructively negate a predicate. In other words, it is not possible to have the CLP engine construct a term that deliberately *fails* to satisfy a given predicate. Given a predicate p, we can query $\exists x.p(x)$ but we cannot query $\exists x.\neg p(x)$. While CLP provides the related notion of "negation-as-failure", this operation is not constructive; negation-as-failure cannot construct terms, only filter out unsatisfying terms [101, 102]. In order to get the effect of constructively negating a predicate p, we must create a new predicate $\hat{p}$ that constructively describes the negation of p. This new predicate will contain redundant code, and the resulting specifications are longer and more confusing. Attempts have been made to solve this problem (e.g., [103]), but those solutions require specialized implementations and still require additional code and effort.

4.3 Finding Typechecker Bugs

A language's type system provides guarantees about program behavior, i.e., it excludes behaviors that the language developers have deemed "bad". [104] The type system is essentially a logical theory by which the typechecker attempts to prove that a program does not exhibit these bad behaviors. One can think of the typechecker as a filter which allows through all programs that it can guarantee are well-behaved, while forbidding all programs that it cannot guarantee are well-behaved.

Because exactly determining which programs are well-behaved or ill-behaved is provably undecidable, the typechecker will conservatively reject some potentially well-behaved programs; the fewer such programs it rejects, the more *precise* the typechecker is. However, the typechecker should never accept any program that is potentially ill-behaved; this requirement is called *soundness*. In addition, the typechecker should be *consistent* in its decisions to avoid programmer confusion: similar programs from a typing perspective

should all be accepted or rejected similarly.

In this section we describe methods and techniques for detecting bugs in a typechecker implementation. We focus specifically on **precision bugs**, **soundness bugs**, and **consistency bugs**, though at the end we also discuss a few other kinds of bugs that we encounter as a side-effect of our main focus. Determining what exactly constitutes a bug requires a specification to compare against. A formal specification would be best (for typecheckers, this would be a formal type system) but is not always available, especially for a language under rapid development. In the absence of a formal specification, we rely on an informal notion of "developer intent", gleaned from discussions with the language developers themselves. When we say "spec" below, we are referring to either the formal or informal specification, whichever is available. We do not consider the problem of determining whether the spec itself is correct (e.g., proving the soundness of the type system), though that is an interesting problem to tackle in the future.

4.3.1 Finding Precision Bugs

We wish to automatically generate programs that expose precision bugs in the typechecker. We first need a definition that tells us when a program exposes a precision bug:

Definition 1 ⟨***Precision Bug***⟩*: A program exposes a typechecker precision bug if the program is well-typed according to the spec but the typechecker rejects the program.*

Because the spec can be informally defined, it may be uncertain whether the program is well-typed according to the spec. Even if we can guarantee that the program is well-behaved, the type system implemented by the typechecker may not be able to prove that fact and in that case the program *should* be rejected by the typechecker. However, we

can give a more specific condition under which the program should probably have been accepted:

Corollary *A program exposes a precision bug if the typechecker has computed information that implies the program is well-typed, but rejects the program anyway.*

For example, suppose that the typechecker can infer the program is well-typed if it can prove some proposition q. It has already proved proposition p, and it knows that $p \implies q$. Thus, the typechecker should be able to derive q and declare the program well-typed. However, if it ignores that information and rejects the program then we say it has a precision bug.

From these definitions, it suffices to generate well-typed programs using the technique described in Section 4.2, run them through the typechecker, and see whether the typechecker accepts them or not. If a program is rejected, then given a formal spec we are guaranteed that we have exposed a bug. Given an informal spec, we have exposed a case where either there is a bug in the typechecker or the language developers need to tweak their notion of well-typedness to refine the informal spec.

4.3.2 Finding Soundness Bugs

We wish to automatically generate programs that expose soundness bugs in the typechecker. We first need a definition that tells us when a program exposes a soundness bug:

Definition 2 ⟨***Soundness Bug***⟩*: A program exposes a typechecker soundness bug if the program is not well-typed according to the spec but the typechecker accepts it as valid.*

Thus, in order to expose soundness bugs we must generate ill-typed programs. In concept, this is trivial: simply generate syntactically valid programs and filter out all

those that are well-typed (as the generated programs grow larger, the odds of a syntactically well-formed program also being well-typed tend to shrink exponentially). However, the resulting ill-typed programs are generally *obviously* ill-typed, such that even a buggy typechecker would probably be able to correctly reject them. Intuitively, we want the ill-typed programs to be *non-obvious* so that even a mostly-correct typechecker might still trip up and incorrectly accept them.

For this purpose, we introduce the notion of "*almost* well-typed" programs. The idea is simple: given a set of type system rules, we pick a subset of the rules' premises and negate them. Any program that is well-typed according to the modified type system is *almost* well-typed according to the original type system—that is, the program is ill-typed, but in a precisely controlled way. This notion is independent of the particular type system that the typechecker implements, allows us to tune the degree of ill-typedness at a fine granularity (by choosing how many and which premises to negate), and is intended to mirror likely mistakes that might be made when implementing the typechecker (for example, forgetting to check a rule's premise or checking it incorrectly).

Example: Almost Well-Typed System F

```
1  typeof(Gamma, app(E1, E2), T2) :-
2       typeof(Gamma, E1, arrow(T1, T2)),
3       typeof(Gamma, E2, T3),
4       T1 \== T3.
```

Figure 4.4: CLP specification of *almost* well-typed System F. The only change is to the clause for the APP rule, the rest of the clauses are the same as Figure 4.3 and are omitted here. The `\==` operator succeeds if its two operands are non-equal; in the context where it is used, it stipulates that `T1` and `T3` must be distinct types, the exact *opposite* of the original rule.

We illustrate this idea using the System F example from Section 4.2. Consider Fig-

ure 4.2, which gives the typing rules for our variant of System F. Suppose that we decide to negate the second premise of the APP rule, which says $\Gamma \vdash e_2 : \tau_1$ (i.e., that the type of the argument matches the type of the function's parameter). Negating this premise means ensuring that the type of the argument e_2 is *not* the type of the function parameter τ_1. Figure 4.4 gives a modified implementation of the APP rule using CLP, which can be compared to the CLP implementation given in Figure 4.3. The clause in Figure 4.4 is generating a type `T3` for `E2` and then ensuring that `T3` is *not* the same as `T1`. We would prefer to use constructive negation to create a type `T3` that is different from `T1` by construction, however as discussed in Section 4.2.2 this is not possible in typical CLP languages. Thus, the almost well-typed implementation must spell out to the CLP engine what negation means in the context of each negated premise.

While this example requires that *every* function application in the generated program is ill-typed, we can also specify that only a certain *number* of function applications are ill-typed, e.g., that there is exactly one ill-typed function application and all the rest are well-typed. In general, for any *almost* well-typed program generation we can specify how many times each negated premise is used versus the original premise. We accomplish this by adding a counter that counts the number of times a negated premise is applied; if the counter exceeds some bound then the negated premise cannot be used anymore and the generator must use the original, non-negated premise.

4.3.3 Finding Consistency Bugs

We wish to automatically generate programs that expose consistency bugs in the typechecker. We first need a definition that tells us when two programs expose a consistency bug:

Definition 3 ⟨***Consistency Bug***⟩*: Two programs together expose a typechecker consistency bug if (1) the well-typedness (ill-typedness) of one implies the well-typedness (ill-typedness) of the other; and (2) the typechecker accepts one program and rejects the other.*

To find consistency bugs according to this definition, we need a method to generate "type-equivalent" programs—that is, programs that satisfy point (1) in Definition 3. Generally, type-equivalence is specific to the language under test. For expository reasons, we provide a simple example to illustrate the general idea. Consider the following program:

$$\texttt{let x:}\tau\texttt{ = }e_1\texttt{ in }e_2$$

In the program above, τ is some arbitrary type and e_1 and e_2 are arbitrary expressions. The overall meaning of this program is to evaluate e_1 down to a value of type τ, assign the result to the variable `x`, and then evaluate e_2 with `x` in scope. With this simple setup, we can automatically transform this program into the equivalent one below:

$$\texttt{let t:}\tau\texttt{ = }e_1\texttt{ in (let x:}\tau\texttt{ = t in }e_2\texttt{)}$$

The program above preserves the meaning and typedness (whether well-typed or ill-typed) of the original program, even without knowing exactly which result it is. If one of the above programs is accepted by a typechecker and the other is rejected, then the typechecker has a consistency bug.

While consistency bugs are theoretically redundant with soundness bugs, explicitly checking for them offers some distinct advantages. For one, this focuses in on a very specific portion of the typechecker state space, specifically the parts of the state space where consistency bugs can exist. Historically, such focusing has helped to reveal additional bugs not detected by an unfocused technique (e.g., prior work on *swarm testing* [105, 106, 18]). Additionally, consistency bugs are useful in a context where complete

knowledge of the underlying type system is unavailable, which is true in practice for informal type systems like that of Rust. Without complete knowledge, we cannot possibly generate all possible ill-typed programs because we simply do not know if a completely arbitrary program is ill-typed. However, consistency bugs can still allow us to test features without complete knowlege; consistency bugs require knowing only that two programs behave similarly, without requiring us to define exactly what that behavior is.

In order to systematically check for consistency bugs, we first generate well-typed and almost well-typed programs according to the methods described in Sections 4.3.1 and 4.3.2, then apply a series of language-specific transformations on the resulting programs to create type-equivalent sets of programs, then run each type equivalence class of programs through the typechecker to see whether they are all accepted or rejected.

4.3.4 Other Kinds of Bugs

While our main focus is on precision, soundness, and consistency bugs, in the process of finding them we can encounter other kinds of bugs as well. Two common kinds of bugs that we may encounter are **parser bugs** and **crash bugs**, as defined below.

Definition 4 ⟨***Parser Bug***⟩*: A program exposes a parser bug if it is syntactically well-formed according to the spec but the parser rejects it.*

Definition 5 ⟨***Crash Bug***⟩*: A program exposes a crash bug if it causes the typechecker to crash when typechecking it.*

Not all bugs neatly fit into the aforementioned categories. We call these **miscellaneous bugs**, as defined below:

Definition 6 ⟨***Miscellaneous Bug***⟩*: A program exposes a bug which is not clearly identifiable as a* ***precision***, ***soundness***, ***consistency***, ***parser***, *or* ***crash*** *bug.*

We do not do anything special to find these three additional kinds of bugs, but merely make a note when our testing encounters them.

4.4 Testing the Rust Typechecker

In this section we describe Mozilla's Rust language [71], with particular attention to its type system, as well as a set of program generators that we have implemented for the Rust typechecker using the techniques described in Section 4.3. Rust serves as an interesting case study for these techniques because it is under active development and features a sophisticated but informally-defined type system. The lack of formal specification means that we cannot establish ground truth regarding typedness, but must instead establish a dialogue with the Rust developers to evaluate the results of our testing. This situation is common in large-scale, industrial-strength language development, and our successful application of these techniques to Rust demonstrates that they can handle such languages. This work is the first to successfully generate well-typed Rust programs and to systematically test the Rust typechecker. All of the program generators described here are available in this chapter's supplementary materials[1].

4.4.1 Rust Background

Rust is intended to be a systems-level programming language along the lines of C and C++, but with much greater safety guarantees afforded by its type system. Rust supports tuples, records, generics, parametric polymorphism, type classes, associated types, affine types, and borrowing. We briefly describe some of the less common typing features: type classes, associated types, affine types, and borrowing.

[1] `http://www.cs.ucsb.edu/~kyledewey/ase15.zip`

Type Classes

First introduced in Haskell, type classes [107] provide a way of allowing for ad-hoc type polymorphism in a manner which is more principled than that of typical object-oriented programming. A type class declares a set of polymorphic function signatures that must be implemented by all members of that class. Polymorphic type variables can then be constrained to require that they belong to a given type class. Type classes are interesting from a well-typed program generation standpoint because determining well-typedness requires reasoning about type constraints arising from an intricate mixture of syntactic and semantic features.

Associated Types

A useful feature seen in Standard ML, C++, Haskell, and Rust, among others, is that of associated types [108, 109, 110], which are intended to simplify polymorphic code. This feature allows auxiliary type variables to be associated with some type τ, such that these auxiliary variables are *implicitly* passed whenever τ is *explicitly* passed. In practice, this feature can dramatically cut down on the number of type variables which must explicitly be passed in the code, greatly reducing boilerplate.

Affine Types

One of the most recognized features of Rust is its use of affine types (closely related to linear types [111, 112]) over memory regions [113]. Rust did not pioneer the use of affine types for this purpose (see, e.g., [114, 115], among others), but it is the first language to use them that has substantial industry support. Rust uses affine types for automated memory management without garbage collection or reference counting. By default, all variables use affine types (hereafter referred to as "affine variables"). The

key property that affine types enforce is that any affine variable cannot be used more than once. Intuitively, this makes the values of affine variables finite resources, and accessing an affine variable consumes this resource. If an affine variable goes out of scope without having its value be used, then the underlying memory for that value can be safely reclaimed. Crucially, the information regarding when this reclamation can safely occur is known entirely at compile time, so the compiler can automatically inject `free`-like operations into the code. In fact, programmers usually cannot call `free` themselves; this is considered entirely the job of the compiler.

To better illustrate how affine types work in Rust, consider the following *ill-typed* Rust code:

```
fn dup1<A, B>(a:A, b:B) -> (A, A) { (a, a) }
```

The above code declares a polymorphic function `dup1` with two parameters `a` and `b` whose return type is a tuple with elements the same type as parameter `a`. This code is ill-typed because `a` is used twice in the body of the function to construct the pair being returned. The following modified version is well-typed:

```
fn dup2<A, B>(a1:A, a2:A, b:B) -> (A, A) { (a1, a2) }
```

In the code above, the values of parameters `a1` and `a2` are consumed to produce the return value, while parameter `b` is unused and thus its value is unconsumed. Therefore, `b`'s value will be automatically reclaimed when `dup2` returns (i.e., the compiler injects a hidden `free(b)` call at the end of `dup2`).

Borrowing

Affine types are severely restrictive in practice, as shown in the previous `dup1` example where it was not possible to duplicate the parameter `a`. To alleviate this problem, Rust relaxes the affine restrictions in a sound manner by using references. Intuitively, when

the programmer creates a reference to a value, that reference *borrows* the value for a clearly-defined duration without consuming it. The borrow durations are made explicit in the type system via *lifetime variables*, which are associated with every reference and constrain how long a reference is permitted to borrow a value.

To demonstrate how borrowing works, consider the following well-typed Rust program:

```
fn dupref<'a,A>(r: &'a A) -> (&'a A, &'a A) { (r, r) }

fn calldup<A>(a: A) {
  let r = &a;
  let (d1, d2) = dupref(r);
}
```

In the above code, `'a` is a lifetime variable. The `dupref` parameter `r` is a reference to a type `A` with lifetime `'a`. Unlike the data referenced, the reference itself can be treated nonlinearly, as shown by the return value which uses `r` twice to construct a tuple. Lifetime variables are created automatically by the compiler, as shown in `calldup` which creates the initial reference to `a`. The typechecker is responsible for verifying that reference lifetimes are properly observed to avoid memory safety violations.

4.4.2 Rust Program Generators

Rather than create a single program generator that attempts to encompass all of Rust at once, we create separate generators which focus in on different parts of Rust's type system, similar in spirit to *swarm testing* [105]. These generators are overall much more complex than the example shown in Section 4.2.1, with each spanning several hundred lines of code. The following discussion is intended to provide the overall gist of how these generators work; the complete generators are available in this chapter's supplementary

materials[2].

Generator G1

This generator creates programs with well-typed first-order functions and function bodies. This generator handles memory regions, lifetime variables, first-order function calls, loops, variables, conditionals, and references. Because lifetime variables model the duration under which a reference is valid, we treat them as symbolic integers. This representation is amenable to CLP's built-in arithmetic constraint solvers. Overall, this feature set represents the very heart of Rust and typical programs would use this portion most frequently. This generator demonstrates CLP's capability to handle the core features of Rust's type system. Unfortunately, G1 is occasionally less precise than Rust's typechecker in ways which are difficult to address without augmenting G1 with a dataflow analysis. As such, we did not experiment with generating *almost* well-typed programs from G1, as such programs may still be well-typed in Rust thanks to additional information G1 does not track.

Generator $\mathbf{G2_W}$

This generator creates programs with well-typed records, typeclasses, and typeclass implementations. This generator internally implements a simple, specialized constraint solver over type constraints, which mirrors a similar constraint solver in Rust's typechecker implementation [116]. A sanitized snippet from our constraint solver implementation is shown in Figure 4.5, which has been stripped of code related to bounded-exhaustive search and has undergone some variable and procedure renaming. Overall, CLP is highly amenable to this approach of implementing custom constraint solvers, a fact which has been noted elsewhere [117].

[2] `http://www.cs.ucsb.edu/~kyledewey/ase15.zip`

```
1  constraintHolds(State1, LifetimeCons, FinalState) :-
2    LifetimeCons = lifetimeCons(Lifetime1, Lifetime2),
3    lifetimeInScope(State1, Lifetime1),
4    lifetimeInScope(State1, Lifetime2),
5    ensureLivesAtLeast(Lifetime1, Lifetime2),
6    addAssumption(State1, LifetimeCons, FinalState).
7  constraintHolds(Stat1, TypeLifetimeCons, FinalState) :-
8    TypeLifetimeCons = typeLifetimeCons(Type, Lifetime),
9    lifetimeInScope(State1, Lifetime),
10   inhabitedType(State1, Type, State2),
11   addAssuption(State2, TypeLifetimeCons, State3),
12   handleImpliedLifetimeConstraints(
13     State3, Type, Lifetime, FinalState).
```

Figure 4.5: Snippet of sanitized code handling two of the three possible Rust type constraints we consider, which is used as a constraint solver. The first case of `constraintHolds` (starting on line 1) handles the Rust constraint `'lt1 : 'lt2`, meaning the lifetime `'lt1` lives at least as long as the lifetime `'lt2`. The second case (starting on line 7)handles the constraint `Type : 'lt`, meaning the type `Type` lives at least as long as the lifetime `'lt`. Descriptions of select data involved and called procedures is provided inline below.
`State` Holds everything in scope, including lifetime variables, type variables, typeclasses, typeclass implementations, and previous typing assumptions made. Different actions manipulate the state, resulting in new states.
`lifetimeInScope(S,L)` succeeds if the lifetime variable `L` is in scope with respect to state `S`.
`ensureLivesAtLeast(L1,L2)` succeeds if lifetime `L1` lives at least as long as `L2`, potentially adding a CLP arithmetic constraint roughly of the form `L1` $\leq$ `L2`, where smaller values represent longer-living lifetimes.
`addAssumption(S1,C,S2)` records that the constraint `C` has been assumed to be true. `C` is added to state `S1` to yield state `S2`.
`inhabitedType(S1,T,S2)` succeeds if the type `T` is inhabited under state `S1`, yielding state `S2`. Depending on `T`, `constraintHolds` may end up being called in a mutually recursive fashion, as determining if an arbitrary type is inhabited in Rust entails checking constraints on types. New information can result from these recursive calls, hence the need for state `S2`.
`handleImpliedLifetimeConstraints(S1,T,L,S2)` records any information implied by the fact that the type `T` has been shown to live at least as long as lifetime `L`. For example, if `T` is `&'a Foo` and `L` is `'b`, then we also know that `Foo : 'a` and `'a : 'b`. New information is added to state `S1`, yielding state `S2`.

Generator $\mathbf{G2_I}$

We used our technique for *almost* well-typed program generation (see Section 4.3.2) and modified the $\mathrm{G2_W}$ generator to create ill-typed programs for the same subset of Rust. This was achieved by nondeterministically skipping calls to the `constraintHolds` procedure shown in Figure 4.5, leading to the introduction of constraints with unverified validity. Because a constraint may still hold by chance, an additional check was performed at the end of generation to ensure that the program was not accidentally well-typed, much like the check performed in Figure 4.4.

Generator G3

This generator also creates programs with records, type classes, and type class implementations, but in a much less constrained way than generator G2. It handles a larger subset of Rust, including associated types, but does not guarantee that the generated programs are well-typed and neither does it guarantee that they are *almost* well-typed. Instead, it uses the techniques described in Section 4.3.3 to create type-equivalent classes of programs. The purpose of this generator is to find consistency bugs, and so it does not matter whether the programs are well-typed or not—only that the typechecker treats them consistently. Our program transformations to create type-equivalent programs mainly move the placement of explicit type constraints in the generated code such that the movement should have no effect on typedness—the concept is analogous to replacing $A \wedge B$ with $B \wedge A$. While there is overlap between this generator and others, G3 is specifically targeted to find consistency bugs with high likelihood.

Generator $\mathbf{G4_W}$

This generator is similar to $G2_W$ but the internal constraint solver is even more precise, enabling the generator to recognize more programs as being well-typed. This internal constraint solver takes more implied type information into account than in $G2_W$, and is generally even more precise than the constraint solver implemented in the Rust typechecker. This serves to point out places where the Rust typechecker needlessly loses precision. This is actually a defective early prototype of $G2_W$, and bugs found by $G4_W$ were used to help inform how Rust's typechecker works. Because the behavior of $G4_W$ intentionally differs from Rust, it quickly finds many possibly duplicate bugs, necessitating triage. As a result, there are potentially more issues to be found with $G4_W$ than what we report.

Generator $\mathbf{G4_I}$

We used our technique for *almost* well-typed program generation (see Section 4.3.2) and modified the $G4_W$ generator to create ill-typed programs for the same subset of Rust. This was done in the exact same way as with the creation of $G2_I$, by nondeterministically skipping calls to the `constraintHolds` procedure. As with $G4_W$, this quickly finds many issues, and so this generator may have actually found more distinct issues than what we report.

4.5 Evaluation

We evaluate our techniques for typechecker fuzzing by implementing the program generators described in Section 4.4 and applying them to test the Rust language implementation. Through this process we have uncovered 18 bugs according to the definitions given in Section 4.3: 14 that have been acknowledged by the Rust developers, and 4 that

the developers do not wish to consider as bugs. We report the issues we uncovered that the developers have decided not to treat as bugs because (1) they are still bugs according to our definitions; and (2) uncovering these issues often led to in-depth discussion and debate by the developers before deciding that they were not bugs—as such, the questions raised by these issues were useful for tweaking the informal spec even if they did not result in fixes to the language.

4.5.1 Experimental Details

Rust is under active development, thus for our experiments we fixed on testing version 1.0-alpha, which was the most recent version circa the start date of this work. In order to carry out our testing, we implemented a custom SMP parallel fuzzing tool written in a combination of Scala [118] and Rust itself. The Rust portion takes advantage of the fact that Rust compiler internals are exported as a library, allowing us to repeatedly call specifically into the Rust typechecker without incurring the overhead of repeatedly loading in the Rust compiler as a process. Across the entire tool, disk IO occurs only to write out newly discovered bugs; all other interprocess communication occurs through UNIX pipes. Our testing infrastructure and the program generators that we evaluated are all available in this chapter's supplementary materials[3].

We dynamically ensure that duplicate crash bugs are unique using the techniques described in Chen et al. [119]. Automatically detecting duplicate typechecker bugs is an open problem; we did so manually for these experiments, and retroactively modify the generators to avoid repeatedly hitting the same bug where possible. Over a period of over 600 machine-hours we tested nearly 900 million generated programs. In terms of performance, the testing of generated programs turns out to be the bottleneck rather than program generation itself: a single program generation process was always able to

[3] `http://www.cs.ucsb.edu/~kyledewey/ase15.zip`

outpace the testing of the resulting programs (often by orders of magnitude) even when the testing was carried out in parallel threads on a 36-core machine. For example, the G2_W fuzzer generates approximately 138k programs per minute, though we can only test approximately 28k per minute on a 12 core machine. Moreover, the test harness has been heavily optimized, whereas the generators are relatively naive.

4.5.2 Reported Bugs

Tables 4.1 and 4.2 list the bugs that we uncovered during testing. The split between the tables is arbitrary; this was done for space reasons. For each bug we report which program generator was used to find it, its status (whether the developers have confirmed it as a bug or not), and give a brief description of the bug. We discuss the causes and implications of a select few of these bugs below.

Optimistic Treatment of Unsound Expressions

Rust allows for modular code definitions, which impacts typechecking of typeclasses. Specifically, when typeclass implementations are separate from typeclass declarations, there is an issue determining whether or not a given typeclass implementation properly implements its supposed typeclass. Concretely, suppose that the programmer adds a constraint that type `bool` (a built-in primitive type) must implement some user-defined typeclass `TC`. The actual implementation of `TC` for `bool` may not be present in the code being compiled (for example, if that code is a library which will be linked in to other code downstream). This fact raises an ambiguity with respect to the meaning of the constraint, which could be (1) `bool` must immediately implement `TC` in currently available code, or (2) `bool` must *eventually* implement `TC`, including in code that is not yet available. Experimentally, we have determined that the Rust typechecker takes the latter approach,

Bug ID	Kind	Generator	Confirmed?	Brief Description
1	Precision	G1	✗	The type system discards some information around blocks, leading to the rejection of well-behaved programs.
2	Precision	G1	✗	The type system conservatively considers it possible for two references which point to incompatible types to assign into each other.
3	Precision	G1	✗	Forcing lifetime variables to be equivalent triggers precision loss.
4	Precision	G1	✗	In general, it is not possible to refactor code so that an expression is replaced by a call to a function that performs the same operation as the original expression.
5	Precision	$G2_W$	✓	The compiler rejects a program saying that an additional, but irrelevant, constraint is needed.
6	Precision	$G2_W$	✓	Constraints of the form `typ : ...` behave in fundamentally different ways depending on whether or not `typ` is a type variable.
7	Precision	$G2_W$	✓ [120]	Programs which fail to use all lifetime and type variables available in certain syntactic positions are rejected, ultimately to make implementation simpler.
8	Precision	$G4_W$	✓ [121]	The compiler discards implied information in the context of typeclasses.
9	Soundness	$G2_I$	✓ [122]	The typechecker does not check that the type we implement a typeclass for can actually exist.

Table 4.1: Summary of reported issues and bugs, part I. "Confirmed?" is ✓ (developers confirm as a bug), ✗ (developers decided it's *not* a bug). References to Rust language Github issues are provided where possible.

Bug ID	Kind	Generator	Confirmed?	Brief Description
10	Soundness	$G2_I$	✓ [123]	Constraints over lifetime or type variables are discarded if they are unused in certain syntactic positions.
11	Soundness	$G4_I$	✓	The solving of type constraints on typeclass implementations which check that a given type implements a given typeclass are delayed until a typeclass is used.
12	Consistency	G3	✓ [124]	Syntactically duplicating a constraint on a type variable in the type variable position is considered a type error, when logically this is the same as saying $A \wedge A$
13	Consistency	G3	✓ [125]	If two constraints on a type variable in the type variable position refer to the same typeclass with different type parameters, the compiler incorrectly reports the are syntactically identical.
14	Parser	G1	✓ [126]	Fails to parse "`box ()`", which should heap-allocate the unit (`()`) value
15	Crash	$G2_W$	✓ [127]	The solving of constraints involving lifetime parameters are delayed until after they are needed, resulting in internal inconsistencies.
16	Crash	G3	✓ [128]	A crash occurs if a type constraint on a typeclass attempts to refer to its own associated type.
17	Miscellaneous	G1	✓ [129]	Untouched contents of a `struct` can impact typechecking.
18	Miscellaneous	G1	✓	Lifetime variables do not correspond directly to memory regions [113].

Table 4.2: Summary of reported issues and bugs, part II. "Confirmed?" is ✓ (developers confirm as a bug), ✗ (developers decided it's *not* a bug). References to Rust language Github issues are provided where possible.

which is not consistent with the expected behavior from certain bug reports [130], and ultimately leads to bugs 9 and 11 in Tables 4.1 and 4.2, respectively. We label this issue as a soundness bug because the typechecker can optimistically accept unsound code in the vain hope that it will eventually get enough information to prove that the code is sound. The result is that, for example, library code can compile properly, but if a user ever links to that library *and* attempts to use the unsound code, the compiler will unexpectedly reject their code with a very opaque and useless error message. The underlying problem is currently being addressed [122], and ultimately a breaking language change will result.

Lack of Modus Ponens

Because types are propositions, it is possible to use the existence of a type to infer further type constraints using the rules of logic. That is, if we know p and we know $p \implies q$ then we can logically infer q; this is the well-known rule of modus ponens. However, it turns out that Rust does not always apply this rule during typechecking, leading to imprecision and unintuitive behavior. For example, when checking whether a typeclass is properly implemented, the Rust typechecker inconsistently applies modus ponens when checking type constraints, leading to bugs 6 and 8 in Table 4.1. The developers acknowledge that this problem is a fundamental issue, and there are plans to address it [121, 122].

Inconsistent Handling of Type Constraints

Rust allows explicit type constraints to be placed on various language features (e.g., a function definition). To improve flexibility and readability [131], the programmer can place these constraints in either of two syntactic positions. Logically, the syntactic placement of the constraint should have no bearing on its meaning—however, our testing revealed that constraints in different syntactic positions are handled inconsistently (see

bugs 12 and 13 in Table 4.2). The Rust developers have since fixed the underlying problem.

4.6 Conclusions

This chapter has proposed a general technique for using CLP to test the typechecker implementations of statically-typed languages. The particular application to Rust shows that this technique works for complex, informal type systems, as one typically sees in practice. This process allowed us to identify 18 bugs in Rust, of which 14 were confirmed by developers. Considering the popularity and sheer number of users of Rust (with hundreds of millions of related downloads [97]), this is an impressive feat.

In regards to the wider thesis, this case study serves as further evidence of the expressibility and performance of CLP. No one has ever attempted to generate well-typed programs for a type system as complex as that of Rust, so Rust serves to push the boundaries of what is possible to express in CLP. Not only was CLP able to express well-typed and ill-typed Rust programs, it was able to generate such programs much more quickly than they could be tested. Nearly a billion tests were produced overall, demonstrating that CLP has no real limit on the number of tests it can generate. Comparing this work to its closest competitor (namely Fetscher et al. [42]) reveals that CLP is orders of magnitude faster on a significantly more complex language. In short, CLP allowed for greater expressibility than ever before attempted in this space, and it offered orders of magnitude better performance than of its closest competitors in this space. This serves as strong evidence of my overarching thesis, namely that CLP is a viable solution to the structured black-box test case generation problem.

Chapter 5

Case Study: Semantics-Based Fuzzing of SMT Solvers

5.1 Introduction

In this case study, CLP is applied to the problem of generating SMT-LIB [28] formulas whose correct behavior is known ahead of time. This requires deep semantic reasoning about SMT-LIB, again pushing the bounds of what can and cannot be expressed. Moreover, SMT-LIB in and of itself defines an incredibly expressive constraint language, as this is the input language of SMT solvers (described in Chapter 1, Section 1.6.1). As such, reasoning about the behavior of SMT-LIB in CLP requires us to reason about an expressive logic (namely that of SMT-LIB) **in** an expressive logic (namely that of CLP). This clearly pushes CLP to the absolute limit of what is possible in terms of expressibility.

Instead of testing all of SMT-LIB, we instead focus on two subsets: the theory of bitvectors (which features operations like modular arithmetic), and the theory of floating point [132] (which is intended to reason about IEEE-754 floating point numbers [133]). The theory of floating point is incredibly complex, reflecting the general complexity of

floating point numbers. As such, it serves as a particularly good subset of SMT-LIB to push the expressibility limits of CLP. As for testing the theory of bitvectors, the theory of floating point is often translated down into constraints in the theory of bitvectors, motivating testing of the theory of bitvectors itself.

The reason for focusing specifically on the correct behavior of SMT-LIB formulas is motivated by the sorts of applications SMT solvers are used for, such as automated testing (e.g., [6, 7, 8, 10, 134, 135]) synthesis (e.g., [29]), and verification (e.g., [136, 137, 138, 139, 140]). Correctness bugs in SMT solvers can result in incorrect behavior in these applications. This is particularly damaging for the verification domain, which seeks to *prove* software correct; an SMT solver correctness bug in this domain can translate to a faulty proof, meaning the underlying software is unproven. Such a situation defeats the entire purpose of verification, so it is desirable to find these sorts of bugs in SMT solvers *before* they become problems.

There is a variety of existing work related to testing various constraint solvers, notably SAT solvers [141], SMT solvers [17], and Answer Set solvers [142] (which were originally described in Chapter 1, Section 1.6.3). However, none of this work makes any attempt to reason about the actual semantics of the formulas generated. With this in mind, the case study in this chapter breaks new ground.

Related to the overall thesis, this chapter serves as a limit study of the expressibility of CLP. Remarkably, this case study demonstrates that CLP *can* generate such expressive formulas, further backing the claim that CLP is an effective solution to the black-box structured test case generation problem. While the performance of the CLP-based test case generator suffers relative to the previous case studies discussed (with a generation rate on the order of dozens of programs per minute as opposed to the tens of thousands per minute seen in Chapter 2), it still manages to find 23 new bugs across a number of popular SMT solvers, namely Z3 [47] (which has been cited well over 1,000 times [49]),

CVC4 [143], MathSAT5 [144], and Boolector [145]. As such, the test case generator is still fast enough to be effective at finding bugs.

As an aside, this chapter is based on work which was originally submitted to ICSE'17, though was subsequently rejected. Follow-up work has been performed since then which has led to a number of new insights related to exactly how we evaluate the bug-finding effectiveness of a test case generator. Because these insights are unrelated to solving the structured black-box test case generation problem, they will not be discussed further.

5.2 Generating Satisfiable Formulas

This section discusses how to generate SMT formulas which are guaranteed to be *satisfiable*; that is, there exists at least one solution to the input constraints. If a solver under test reports that such an input is *unsatisfiable* (i.e., there exist no possible solutions), it indicates a correctness bug in the solver. Such bugs are considered devastating particularly in the verification domain, so we consider finding this sort of bug a top priority.

We will show how guaranteed satisfiable formulas can be generated in CLP by example. The example we will use is based on the theory of bitvectors, so chosen for its relative simplicity. This choice was purely for expository reasons; the same basic technique also works for the more complex theory of floating point.

The syntax for the subset of the theory of bitvectors we consider in this section is shown in Figure 5.1. Expressions in this language follow a relatively simple type system, described in Figure 5.2. A formal big-step semantics of this language is provided in Figure 5.3, which are crucial for the generation of guaranteed satisfiable formulas.

$$x \in \mathit{Variable} \qquad b \in \{\textsf{true}, \textsf{false}\}$$
$$n \in \mathbb{N} \qquad \vec{b}_n \in \mathit{BitvectorConstant}$$

$$\begin{aligned} \odot \in \mathit{BoolOp} &::= \wedge \mid \vee \\ \otimes \in \mathit{BitvecOp} &::= \textsf{bvadd} \mid \textsf{bvult} \\ e \in \mathit{Exp} &::= x \mid \vec{b}_n \mid b \\ &\quad \mid \textsf{ite}\ e_1\ e_2\ e_3 \mid e_1 \odot e_2 \mid e_1 \otimes e_2 \end{aligned}$$

Figure 5.1: Syntax for the subset of the theory of bitvectors we consider. b represents a Boolean, and $\vec{b}_n$ represents a fixed sequence of boolean values of length n. **bvadd** denotes modular arithmetic, **bvult** denotes unsigned less-than, and **ite** represents an if-then-else construct.

$$n \in \mathbb{N}$$
$$\tau \in \mathit{Type} = \textsf{bool} \mid \textsf{bv}\ n$$
$$\Gamma \in \mathit{Env} = \mathit{Variable} \to \mathit{Type}$$

$$\frac{x \in \texttt{keys}(\Gamma) \qquad \tau \triangleq \Gamma(x)}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{}{\Gamma \vdash \vec{b}_n : \textsf{bv}\ n}\ \text{BVCONST} \qquad \frac{}{\Gamma \vdash b : \textsf{bool}}\ \text{BOOLCONST}$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{ite}\ e_1\ e_2\ e_3 : \tau}\ \text{ITE}$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \textsf{bool}}{\Gamma \vdash e_1 \odot e_2 : \textsf{bool}}\ \text{BOOLOP} \qquad \frac{\Gamma \vdash e_1 : \textsf{bv}\ n \qquad \Gamma \vdash e_2 : \textsf{bv}\ n}{\Gamma \vdash e_1 \otimes e_2 : \textsf{bv}\ n}\ \text{BVOP}$$

Figure 5.2: Type system for the subset of the theory of bitvectors defined in Figure 5.1. **bool** is a Boolean type, and **bv** n is a type of a bitvector containing a fixed number of bits n.

For our purposes, queries to an SMT solver consist of two components with a particular behavior:

1. A number of variable definitions, along with the types of the variables involved.

   With respect to the semantics in Figure 5.3, these variable definitions ultimately

$$v \in \mathit{Value} = b + \vec{b}_n$$
$$\sigma \in \mathit{Store} = \mathit{Variable} \to \mathit{Value}$$
$$\Downarrow \in \mathit{eval} = (\mathit{Store} \times \mathit{Exp}) \to \mathit{Value}$$

$$\frac{x \in \texttt{domain}(\sigma) \qquad v \triangleq \sigma(x)}{\sigma \cdot x \Downarrow v}\ \text{VAR} \qquad \frac{}{\sigma \cdot \vec{b}_n \Downarrow \vec{b}_n}\ \text{BVCONST} \qquad \frac{}{\sigma \cdot b \Downarrow b}\ \text{BOOLCONST}$$

$$\frac{\sigma \cdot e_1 \Downarrow \textsf{true} \qquad \sigma \cdot e_2 \Downarrow b_2 \qquad \sigma \cdot e_3 \Downarrow b_3}{\sigma \cdot \textsf{ite}\ e_1\, e_2\, e_3 \Downarrow b_2}\ \text{ITE-B-T}$$

$$\frac{\sigma \cdot e_1 \Downarrow \textsf{false} \qquad \sigma \cdot e_2 \Downarrow b_2 \qquad \sigma \cdot e_3 \Downarrow b_3}{\sigma \cdot \textsf{ite}\ e_1\, e_2\, e_3 \Downarrow b_3}\ \text{ITE-B-F}$$

$$\frac{\sigma \cdot e_1 \Downarrow \textsf{true} \qquad \sigma \cdot e_2 \Downarrow \vec{b}_{n2} \qquad \sigma \cdot e_3 \Downarrow \vec{b}_{n3}}{\sigma \cdot \textsf{ite}\ e_1\, e_2\, e_3 \Downarrow \vec{b}_{n2}}\ \text{ITE-BV-T}$$

$$\frac{\sigma \cdot e_1 \Downarrow \textsf{false} \qquad \sigma \cdot e_2 \Downarrow \vec{b}_{n2} \qquad \sigma \cdot e_3 \Downarrow \vec{b}_{n3}}{\sigma \cdot \textsf{ite}\ e_1\, e_2\, e_3 \Downarrow \vec{b}_{n3}}\ \text{ITE-BV-F}$$

$$\frac{\sigma \cdot e_1 \Downarrow b_1 \qquad \sigma \cdot e_2 \Downarrow b_2 \qquad b_3 \triangleq b_1 \mathbin{\overline{\wedge}} b_2}{\sigma \cdot e_1 \wedge e_2 \Downarrow b_3}\ \text{AND}$$

$$\frac{\sigma \cdot e_1 \Downarrow b_1 \qquad \sigma \cdot e_2 \Downarrow b_2 \qquad b_3 \triangleq b_1 \mathbin{\overline{\vee}} b_2}{\sigma \cdot e_1 \vee e_2 \Downarrow b_3}\ \text{OR}$$

$$\frac{\sigma \cdot e_1 \Downarrow \vec{b}_{n1} \qquad \sigma \cdot e_2 \Downarrow \vec{b}_{n2} \qquad \vec{b}_{n3} \triangleq \vec{b}_{n1} \mathbin{\overline{+}} \vec{b}_{n2}}{\sigma \cdot e_1\ \textsf{bvadd}\ e_2 \Downarrow \vec{b}_{n3}}\ \text{BVADD}$$

$$\frac{\sigma \cdot e_1 \Downarrow \vec{b}_{n1} \qquad \sigma \cdot e_2 \Downarrow \vec{b}_{n2} \qquad b \triangleq \vec{b}_{n1} \mathbin{\overline{<}} \vec{b}_{n2}}{\sigma \cdot e_1\ \textsf{bvult}\ e_2 \Downarrow b}\ \text{BVULT}$$

Figure 5.3: Formal big-step semantics for the subset of the theory of bitvectors covered by the syntax provided in Figure 5.1. The $\overline{\wedge}$ and $\overline{\vee}$ operators represent Boolean AND and OR at the metalanguage level, with the usual definitions. The $\overline{+}$ operation represents modular arithmetic at the metalanguage level, with the usual definition. The $\overline{<}$ operation represents unsigned arithmetic less-than at the metalanguage level, with the usual definition.

determine what the domain of *Store* is.

2. A single expression which may use the aforementioned variable definitions an arbitrary number of times. If the expression is satisfiable, meaning it can be **true**, the solver returns a particular *Store* (σ) under which this is true. If the expression is unsatisfiable, meaning it is **false** for all possible instances of *Store*, then the solver reports the special *unsat*.

With this information in tow, we can provide a formal definition of satisfiability with respect to the *eval* ($\Downarrow$) function defined in Figure 5.3:

Definition 7 *Satisfiability A formula e is* satisfiable *iff* $\exists\sigma \,.\, \sigma \Downarrow e \rightarrow \textbf{\textit{true}}$

A key observation for automatically deriving satisfiable formulas hinges on the very definition of *eval* itself. With the help of CLP, we can use *eval* in an operational manner to effectively execute "backwards"; that is, producing satisfiable formulas given the information that the formula must evaluate to `true`. With this in mind, we can implement the rules in Figure 5.3 using CLP, and then use a modern CLP engine to generate satsifiable formulas from these rules. An example CLP implementation of these rules is shown in Figure 5.4, along with a query which can be used to produce satisfiable formuals. Note that this example is missing information pertaining to bounding the state space; a more complete discussion of this problem and its solutions are in Chapters 7 (particularly Section 7.3) and 9.

In Figure 5.4, the definitions of `boolAnd`, `boolOr`, `add`, and `lessThan` are omitted. Clearly, these operations are crucial to these semantics, and there are multiple approaches for implementing these. One approach is to implement these entirely in pure Prolog; for example, to define `add` directly over lists of bits, and to implement something like a ripple-carry adder. This is arguably the most straightforward approach, though it is quite

```
1  % Store, expression, value
2  eval(Store, var(X), Value) :- lookup(Store, X,  Value).
3  eval(_, bvconst(BN), bv(BN)).
4  eval(_, boolconst(B), bool(B)).
5  eval(Store, ite(E1, E2, E3), bool(B2)) :-
6    eval(Store, E1, bool(true)),
7    eval(Store, E2, bool(B2)), eval(Store, E3, bool(_)).
8  eval(Store, ite(E1, E2, E3), bool(B3)) :-
9    eval(Store, E1, bool(false)),
10   eval(Store, E2, bool(_)), eval(Store, E3, bool(B3)).
11 eval(Store, ite(E1, E2, E3), bv(BN2)) :-
12   eval(Store, E1, bool(true)),
13   eval(Store, E2, bv(BN2)), eval(Store, E3, bv(_)).
14 eval(Store, ite(E1, E2, E3), bv(BN3)) :-
15   eval(Store, E1, bool(false)),
16   eval(Store, E2, bv(_)), eval(Store, E3, bv(BN3)).
17 eval(Store, and(E1, E2), bool(B3)) :-
18   eval(Store, E1, bool(B1)), eval(Store, E2, bool(B2)),
19   boolAnd(B1, B2, B3).
20 eval(Store, or(E1, E2), bool(B3)) :-
21   eval(Store, E1, bool(B1)), eval(Store, E2, bool(B2)),
22   boolOr(B1, B2, B3).
23 eval(Store, bvadd(E1, E2), bv(BN3)) :-
24   eval(Store, E1, bv(BN1)), eval(Store, E2, bv(BN2)),
25   add(BN1, BN2, BN3).
26 eval(Store, bvult(E1, E2), bool(B)) :-
27   eval(Store, E1, bv(BN1)), eval(Store, E2, bv(BN2)),
28   lessThan(BN1, BN2, B).
29
30 % Query for guaranteed satisfiable generation
31 ?- createStoreTemplate(Store),
32    eval(Store, Exp, bool(true)).
```

Figure 5.4: CLP implementation of the rules defined in Figure 5.3, along with a query which can be used to generate guaranteed satisfiable formulas in this language. The `boolAnd`, `boolOr`, `add`, and `lessThan` operations correspond to $\overline{\wedge}$, $\overline{\vee}$, $\overline{+}$, and $\overline{<}$, respectively. `lookup` is a standard map lookup routine. `createStoreTemplate` creates a store where only the variables are instantiated, not their particular values. Effectively, `createStoreTemplate` puts a number of variables in scope, which may or may not be utilized in `eval` via the `var` rule.

inefficient in practice. The underlying reason why is because it can force decisions to be made before all information is available. For example, consider the following program:

```
x bvult (y bvadd z)
```

Ultimately, the above snippet asks whether or not there is some value x which is less than $y + z$. Without any other constraints in play, this is clearly true for some possible assigments of x, y, and z. However, with a pure Prolog approach, it is possible that the value of x will be chosen before even considering y and z. If x was chosen to be the largest possible value representable given the number of bits in play, then this will fruitlessly explore all possible combinations of values for y and z before discovering that no assignment to x and y makes this satisfiable.

Ultimately, the problem with the pure Prolog approach is that the engine explores choices in a poor ordering for the problem. Addressing this problem in general naturally leads to a need for a constraint solver. In this way, there is a catch-22: we ultimately want to test constraint solvers (namely, SMT solvers), but in order to do this efficiently, we need constraint solvers. This clearly leads to vacuous testing if we use the exact same solver in the exact same configuration under test in order to solve this problem. However, we observe that in practice, it is usually possible to avoid this sort of vacuous testing. One approach is to use constraint solvers directly available in most CLP engines, which are quite different from those employed in SMT solvers. Another approach is to simulate operations under test using a different theory. For example, it is possible to encode the constraints of our $\overline{+}$ operation using only integer constraints, which are completely separate from the bitvector constraints we aim to test.

5.3 Generating Unsatisfiable Formulas

Section 5.2 thoroughly discusses catching cases where the solver incorrectly returns `unsat` on satisfiable inputs. This section discusses the opposite situation, where a solver returns `sat` on an unsatisfiable input. In a verification setting, this sort of bug can prevent correct theorems from being proven. In a concolic execution setting (e.g., [6]), this sort of bug can lead to bogus test inputs which do not actually expore the paths they suggest. From the standpoint of the SMT solver, these sort of bugs can represent major semantic misunderstandings. It is relatively easy to double-check an answer using a relatively simplistic method once a solver produces it, and the popular SMT solvers Z3 [47] and CVC4 [143] both have buit-in support for this capability. With this in mind, these sort of bugs will usually be caught internally by the solver, leading to assertion violations (i.e., crash bugs) as opposed to correctness bugs. However, if the double-check fails, this means that *both* the solver itself and the simplistic check are faulty in likely the exact same way, suggesting that a fundamentally incorrect semantics has been implemented.

In order to find these sort of bugs, it is necessary to produce formulas which are guaranteed to be unsatisfiable. We can formally specify what this means with respect to the `eval` ($\Downarrow$) function defined in Figure 5.3, leading to the following definition:

Definition 8 *Unsatisfiability A formula e is* unsatisfiable *iff* $\forall \sigma \,.\, \sigma \Downarrow e \rightarrow \textsf{\textbf{false}}$

As with our definition of satisfiability (Definition 7), this is a relatively standard definition. As before, we could immediately use this to generate guaranteed unsatisfiable programs given a naive generator of expressions e and stores σ. Once again however, this is impractically slow, even if we constrained the generation to produce only well-typed formulas. Moreover, there is no guarantee that the sort of formulas generated are at all *interesting* from the standpoint of the solver. That is, intuitively, we would like to generate formulas which are unsatisfiable, but not *obviously* unsatisfiable, e.g., $p \wedge \neg p$ for

some Boolean variable p. The more complex unsatisfiable examples serve as better tests, as the solver must work harder to refute them.

This problem of generating non-obviously unsatisfiable formulas seems similar to the problem of generating *almost* well-typed programs, discussed in Chapter 4, Section 4.3.2. However, such similarities are surprisingly superficial. Taking the same approach as for *almost* well-typed programs, we can selectively negate a premise while *eval* is explored, yielding what should be an unsatisfiable program. However, the resulting program is not guaranteed to be unsatisfiable, but rather *might* be unsatisfiable.

To better understand the source of this problem, consider the example input shown in Figure 5.5. In this example, even if the result of `bvadd` is internally incorrect within `eval` (i.e., a premise was negated), it does not actually have any effect on the correctness of the test overall. This is trivially true because the incorrect result is effectively discarded by `ite`, thanks to the guard always being `true`. As such, we can immediately conclude that in the presence of `ite`, we lose guarantees of unsatisfiable generation.

```
(ite(true, 7, x bvadd y)) bvult 10
```

Figure 5.5: Example showing some of the issues involved in generating guaranteed unsatisfiable formulas. Crucially in this example, the result of `x bvadd y` is unused, independent of exactly what this result is.

The aforementioned problem is, however, not specific to `ite`. Consider the following input:

```
(x bvadd 5) bvult 10
```

The above input will be satisfiable as long as `x bvadd 5` is less than `10`. If `x` was chosen to be `3`, then the expected result is `8`, which is satisfiable. Say, however, we negated the clause in BVADD so that the effective result of `x bvadd 5` is instead `9`. In

this case, even though the result of BVADD was tainted, we still end up with a satisfiable input, which is undesired.

This sort of problem, in general, is due to the presence of disjunctions, both implicit (as with `ite` and `bvult`) and explicit (as with $\vee$). Because the handling of disjunctions is arguably one of the most important capabilities of a constraint solver, it is unacceptable to simply remove them from the space. As such, we must take a different approach. A key insight is that this problem arises because it is not guaranteed that a particular expression e will be tainted even if one of its subexpressions e' is tainted. It is possible to recover this guarantee via the addition of further constraints. To see how this is done, consider again the example in Figure 5.5. This example clearly exhibits the same problem, as the `ite` overall produces the correct result, even though its subexpression `x bvadd y` is faulty. We can concisely fix this problem with an additional constraint which states that the result of the `ite` is different than what is expected if the `bvadd` subexpression returned the correct result. That is, we add constraints which roughly state "the result of this tainted expression matters to the overall result of the formula". In this case, this additional constraint cannot be satisfied, as the `ite` will always return `7`, irrespective of the return value of the `bvadd` expression. Because the additional constraint cannot be satisfied, generation fails, and so this overall expression would not be produced. This failure occurs even before `bvult` is considered. In general, the idea is to add additional constraints to ensure that outer expressions have different values than expected thanks to inner tainted expressions. This process continues all the way up to the toplevel expression, ensuring that the overall result has somehow been affected by the negation of a premise.

5.4 Application to Bitvectors and Floating Point

We applied the general ideas for guaranteed satisfiable and unsatisfiable formula generation in Sections 5.2 and 5.3 to the theories of bitvectors and floating point [132] in SMT-LIB [28]. This section describes key implementation details and challenges we had during this process, starting with the application to the theory of bitvectors.

5.4.1 Application to the Theory of Bitvectors

For guaranteed satisfiable formula generation, our implementation follows a very similar structure to what is provided in Figure 5.4. For the most part, we simply extend the provided rules to add in additional operations available in the real theory of bitvectors. The one discussion point of interest is how exactly we implemented operations like `boolAnd`, `boolOr`, `add`, and `lessThan`, which we had discussed have multiple avenues for implementation. This ended up being a difficult decision to make, and we ended up trying out all three approaches discussed in Section 5.2 A discussion of our experiences with these approaches follows.

We originally went with a pure Prolog approach, wherein bitvectors were represented as lists of Boolean values. While this was very simple and did not rely on auxilliary constraint solvers, we found it to be often impractically slow, even for relatively small inputs. This was especially true for constraints involving negation, as our naive implementation was often forced to perform a brute force search of the state space under these conditions. For this reason, this was unfit for extension to the generation of guaranteed unsatisfiable formulas, as these inherently involve many negated constraints (e.g., an expression should evaluate to any value *except for* the expected value, hence negation). Moreover, the performance problems meant that it was practically necessary to put a timeout on generation in order to ensure practical progress was being made. As such,

while the pure Prolog approach was overall simple, it is not an optimal choice when it comes to generation.

From here, we modified the generator to use the arithmetic constraint solver available in SWI-PL [67, 146], the CLP engine we chose for generation. The idea here was to represent bitvector operations in terms of integer arithmetic operations. For example, `bvadd`'s modular arithmetic can be represented using standard arithmetic addition (specifically constraints like `X #= Y + Z`), along with some additional constraints regarding what happens on overflow. The biggest challenge here was to determine how to encode things using integer constraints, which was still relatively straightforward.

While this approach based on utilizing built-in constraint solvers works in theory, we found it to be fraught with problems in practice, to the point of unusability. For one, while the CLP library supports non-linear operations like multiplication (which greatly simplifies multiplication over bitvectors), we found that the engine was prone to hanging when asked to find solutions for these sorts of constraints. This was true even in contrived situations where a simple brute force search could be completed within milliseconds to find the answer. This required us to manually specify how non-linear operations worked, with was complex and error-prone. Moreover, we found the library to be extremely buggy, to the point where it was more likely for the engine to crash than to actually produce an input. Given that our testing technique relies on a relatively bug-free constraint solver, this quickly made the library in SWI-PL untenable for our purposes.

For these reasons, we ultimately went with the approach of using Z3 [47] to solve these sorts of integer constraints, using custom bindings for SWI-PL [147]. This sped up generation tremendously, and the generated inputs were observed to generally utilize more semantic rules. Z3's robust support for non-linear constraints was also a big win here. While this approach assumes that Z3 ultimately is correct when it comes to solving integer constraints, we found these constraints to be extremely reliable in practice. In

fact, we actually spent nearly a month of CPU time fuzzing Z3's theory of integers using traditional syntactic techniques, which failed to find any bugs. As such, we have high confidence in the correctness of Z3's theory of integers, at least for the sort of queries we were issuing.

5.4.2 Application to the Theory of Floating Point

The complexity of the theory of floating point [132], along with its youth, required us to take a different approach than what we used for the theory of bitvectors. For the theory of floating point, merely getting the semantics correct was a non-trivial problem, which was exacerbated by the fact that no robust solvers yet exist for the theory. As such, when a semantic question arose, we could not simply ask a solver what the correct solution was; the solver could very well be wrong, particularly with the sort of edge cases we were most interested in. This required us to take a different approach, split up into two stages.

The first stage involved writing a naive solver in Scala [118], a functional language. This allowed us to separate out concerns which were specific to generation, which are relevant only in a CLP-based situation. This solver works simply by brute-forcing the entire state space, going through all the possible floating point values within the imposed bounds. For small inputs, this is computationally feasible, and in certain cases it can ironically be even orders of magnitude faster than the solvers under test. Additionally, this still gives us a sound and complete solver, as the state space for the theory of floating point is finite in general; all variables in the theory have a fixed, albeit generally intractable, number of possible values.

Central to our testing technique is the requirement that these semantics be correct. An incorrect semantics leads to inputs which are incorrectly marked as buggy, which

requires human intervention in order to fix. Because inputs can be large, this can require a significant amount of effort. To try to catch as many semantic flaws as possible in our implementation ahead of time, we used a syntactic fuzzing approach to test it against preexisting hardware floating point arithmetic implementations. This ensured that our implementation was at least consistent with existing implementations. In general, this does not guarantee anything — the hardware implementations themselves may be buggy, and there are subtle, intentional differences between the actual IEEE-754 standard [133] and the SMT theory of floating point [132]. In particular, hardware implementations generally can reason only about 32-bit and 64-bit floating point values, whereas the theory of floating point allows an arbitrary (but fixed) number of bits). Additionally, the semantics of edge cases like `min(+0.0, -0.0)` can differ significantly. Even so, this was an effective technique for finding bugs in our own implementation before moving on to using our implementation to find bugs.

Once we were reasonably sure that our Scala-based implementation was correct, we ported this to CLP. This process was usually straightforward, though complexities arose due to the nondeterministic semantics of CLP. For example, the Scala implementation could always assume that only one particular value was in play for a variable at any given point in time, which was the main advantage with going with a brute-force approach. In CLP, it is possible that a variable's value is completely uninstantiated, meaning it nondeterministically holds all possible values at once. As such, in order to assume something in the CLP context, it is necessary to instantiate the variable in a way such that its value reflects whatever assumption is being made. This can get tricky, particularly when it comes to optimizing generation code. Indeed, we found that in practice, if a bug came up in our implementation, it was most likely specific to a generation concern in the CLP implementation, as opposed to a fundamental semantic issue in the Scala implementation.

As for representing bits, we chose the pure Prolog representation of using lists of Boolean values, without he help of any sort of external constraint solver. The reason for this strategy was simplicity: the theory is so complex to begin with that we did not want to introduce any more complexity by adding in auxilliary constraint solvers.

5.5 Evaluation

This section discusses how we applied the formula generators in Section 5.4 to testing a series of SMT solvers. This includes both our overall testing methodology, along with the actual bug-finding results.

5.5.1 Generators Evaluated

The state space of formula generators is significantly larger than just syntactic, guaranteed satisfiable, and guaranteed unsatisfiable. For example, this speaks nothing of the size of the formulas generated, the number of variables they contain, and so on. We informally tried some different combinations which were intuitively likely to find bugs, and we found that the number of bits involved (that is, the number of bits in a bitvector or floating point value) to be significant. We say this was "informal" because we do not have complete evaluation information for all possible parameters; the whole space balloons to over 1,000 unique configurations, which would necessitate approximately 14 months of CPU time to fully evaluate.

Overall, all the test case generators we implemented and evaluated are named and described in Table 5.1. This table is notably missing guaranteed unsatisfiable formula generation for the theory of floating point. Unfortunately, the choice of the pure Prolog approach of representing bits for this theory (discussed in Section 5.4.2) made it impractically slow when used for guaranteed unsatisfiable formula generation. In this context,

Theory	# Bits	Type	Generator Name	Solvers Tested
Bitvectors	3	sat unsat syntactic	**bv_few_sat** **bv_few_unsat** **bv_few_syntactic**	Z3 [47] CVC4 [143] MathSAT5 [144] Boolector [145]
	8	sat unsat syntactic	**bv_many_sat** **bv_many_unsat** **bv_many_syntactic**	
Floating Point	5 (2e + 3m)	sat syntactic	**fp_few_sat** **fp_few_syntactic**	Z3 [47]
	32 (8e + 24m)	sat syntactic	**fp_many_sat** **fp_many_syntactic**	Z3 [47] MathSAT5 [144]

Table 5.1: Implemented test case generators, along with the SMT solvers they were used to test. "sat" means the generator produces guaranteed satisfiable formulas using the technique described in Section 5.2. "unsat" means the generator produces guaranteed unsatisfiable formulas using the technique described in Section 5.3. "syntactic" means the generator produces formula with unknown satisfiability results, using traditional syntactic fuzzing techniques and differential testing [21]. For the floating point configurations, the number of bits (# **Bits**) is broken down into exponent bits ("e") and mantissa bits ("m").

"impractically slow" was on the order of several programs per hour. None of these programs found any bugs, so we simply remove these generators from our evaluation entirely.

Table 5.1 also lists the SMT solvers we test against for each test case generator configuration. Solvers were chosen based on popularity, capabilities (e.g., support for the theory of floating point), and performance. CVC4 [143] and Boolector [145] lack support for the theory of floating point, so we do not test them with our floating point generators. Additionally, while MathSAT5 [144] has support for the theory of floating point, discussion with the authors [148] revealed that the primary focus is on values comprising many bits; bugs found involving few bits were put at lower priority. As such, we only tested MathSAT5 against inputs comprised of many bits.

5.5.2 Testing Process and Infrastructure

As for exactly how these generators were tested, intuitively this follows the following four-step process:

1. Generate a test case
2. Run the test case on a system under test
3. Classify the result from the system under test as being normal or indicative of a bug
4. Report any bugs found along with representative inputs to the appropriate parties

The first two steps were executed in a massively parallel fashion, with a series of identical test case generators producing inputs for a pool of systems under test.

A naive way of performing the second step above is to run the solver under test once per input. However, this is extremely inefficient, with nearly all testing time being spent with the associated costs of building up and tearing down processes. As such, we modified the frontends of each solver slightly so that they can incrementally accept whole new inputs without process teardown, similar to the technique employed in Chapter 4, Section 4.5.1. This improved the testing throughput by up to 10×.

As for the third step, we found that we would often repeatedly hit the same bug, leading to many redundant inputs which would consume progressively more and more disk space. To help alleviate this situation, we implemented a parallel online version of the sort of *fuzzer taming* techniques discussed in Chen et al. [119]. We found that while this was extremely effective for taming assertion violations, it did not apply well to correctness bugs. As such, for correctness bugs, we implemented a custom similarity metric which considered two non-identical formulas to be the equivalent as long as the differences lied

solely in the leaves of their ASTs; this reduced the space of inputs considerably, down to tens of thousands of inputs as opposed to millions of inputs.

With the actual reporting of bug-triggering inputs in step four, while most generated inputs contain no more than a dozen formulas, in some cases these formulas are extremely complex. Based on developer feedback [48], we implemented a delta debugger [75], using the same technique as described in Brummayer et al. 2009 [17]. This significantly cut down on the complexity of reported bug-triggering inputs.

5.5.3 Evaluation Methodology

Ultimately, we want to measure the total number of unique bugs found for each test case generator, particularly the total number of unique *correctness* bugs found. This is somewhat difficult to do because of the large numbers of inputs involved; even after fuzzer taming, we are still left with tens of thousands of inputs, most of which trigger the same bugs. As such, it was necessary to implement some sort of automation in this space.

To achieve this automation, we took a two phase strategy. From a high level, the purpose of the first phase is to discover as many previously unknown bugs as possible, with a significant amount of manual intervention. The goal of the second phase is to automatically rediscover the bugs found in the first phase. Further description of each of these phases follows.

In the first phase, we tested applicable systems using all the test input generators over the course of several months. This was done by incrementally testing and reporting any bugs found. When a bug was found, testing was immediately halted, the bug reported upstream. the revision of the solver was recorded, and the type of bug (either correctness or otherwise) was recorded. Once the bug was fixed upstream, we would record which

revision of the solver fixed the bug, and update our local solver to this latest revsion. From here, testing was restarted. This process was repeated until no new bugs were found.

In the second phase, we tested each of the revisions recorded in the first phase, which includes both the revisions where bugs were originally identified and revisions which fixed the identified bugs. Under the assumption that each revision fixes at most one bug, it is then possible to uniquely identify the bugs found. For example, consider an ordered series of revisions $R_1, R_2, ..., R_n$. Under input I_1, revision R_1 indicates a bug. However, under the same input I_1, revision R_2 does not indicate a bug. From this, we can deduce that R_2 fixes the bug without any manual intervention. Most importantly, if we discover another input I_2 under which R_1 is buggy and R_2 is not buggy, then we know that I_2 is merely another input that triggers the same bug fixed in R_2. As such, we know that only one bug has been uniquely discovered in this example, even though we have found two bug-triggering inputs. Moreover, because we recorded what kind of bug was fixed by R_2 in phase one, we know whether or not this is a correctness bug.

Crucially, the second phase requires no user intervention. As such, it is easy to run each test case generator under its own independent second phase, using a fixed time budget (in our case, ten hours). From this, we can derive exactly both the number of unique bugs and the number of unique *correctness* bugs found by each test case generator.

5.5.4 Results

The results of fuzzing using the above evaluation methodology are shown in Table 5.2. A full discussion of these results follows in Section 5.6. A breakdown of the bugs found on a per-solver basis is shown in Table 5.3. Links to filed bug reports have been included for Z3 and CVC4 in Table 5.3, as these have publically-accessible issue trackers.

Generator Name	**Total Unique Bugs Found**	**Total Unique *Correctness* Bugs Found**
bv_few_syntactic	1	1
bv_few_sat	2	1
bv_few_unsat	0	0
bv_many_syntactic	4	2
bv_many_sat	1	1
bv_many_unsat	0	0
fp_few_syntactic	7	4
fp_few_sat	5	3
fp_many_syntactic	5	3
fp_many_sat	3	3

Table 5.2: The number and types of bugs found in each solver by each generator.

Solver	**Unique Correctness Bugs**	**Unique Total Bugs**	**Bug Report Links**
Boolector [145]	1	1	N/A
MathSAT5 [144]	5	5	N/A
CVC4 [143]	1	4	[149, 150, 151, 152]
Z3 [47]	5	13	[153, 154, 155, 156, 157] [158, 159, 160, 161, 162] [163, 164, 165]
Total	12	23	N/A

Table 5.3: The number and types of bugs found on a per-solver basis. Links to filed bug reports are provided for Z3 and CVC4, which are the only solvers tested with publically-accessible issue trackers.

5.6 Discussion

Our discussion is broken into a quantitative analysis of our results in Table 5.2, along with a qualitative examination of the sort of bugs we found. We start with the quantitative analysis.

5.6.1 Quantitative Analysis

There are three points of interest to take away from Table 5.2, explained thusly.

Unsatisfiable Formula Generation

As shown, generators of guaranteed unsatisfiable formulas uniformly found no bugs. This is due primarily to the fact that the unsatisfiable formula generation is over 6× slower than satisfiable generation, due to the extra constraints involved for unsatisfiable generation (described in Section 5.3). This slow generation rate means far fewer tests are generated, so there are fewer opportunities to hit a bug. Looking at the sort of programs produced, it would appear that the added timeouts (discussed in Section 5.4.1) had the unintended consequence of selecting for low-complexity inputs which could reasonably be generated within the timeout. This biased the state space towards programs which were more easily handled, leading to test inputs which were relatively easy for solvers to get correct.

Guaranteed Satisfiable Formula Generation Often Finds Correctness Bugs

Our technique of generating guaranteed satisfiable formulas finds a number of correctness bugs, and nearly all the bugs it finds are correctness bugs. This shows that it is biased towards testing the semantics of the system under test, as expected.

Syntactic Finds Overall More Bugs

Generally, the syntactic test case generators found more bugs than the generators based on formulas with guaranteed results. The underlying reason why is because the generators with guaranteed results fundamentally explore more narrow state spaces than that of the traditional syntactic generators, so intuitively there are fewer bugs available. This is further substantiated by looking at the sort of bugs found, with the syntactic approaches often finding non-correctness bugs like memory leaks (e.g. [159, 161, 162, 163]). Such bugs have no direct connection to the semantics of SMT-LIB, so the semantics-based

fuzzers are somewhat blind to them.

Of particular interest is that this result seems to contradict that of *swarm testing* [105], which found that focusing on different subsets of a language improves the number of bugs found. We believe that this is due to the fact that SMT solvers comprise a very different domain than those observed in the swarm testing result, one where correctness is absolutely critical. As such, it is somewhat expected that core correctness bugs would be less prevalent, as we observe. Additionally, the effectiveness of swarm testing has only been evaluated in the context of specializing syntax as opposed to specializing semantics as in our approach, so it may be that the swarm testing result is somehow limited to syntax.

5.6.2 Qualitative Examination of Bugs Found

In this subsection, we go through general patterns we found among the bugs discovered, beyond the simple bug counts shown in Table 5.2.

For one, not only were all solvers found to be buggy, all had at least one correctness bug present. This was true both for the young theory of floating point, but also for the well-established theory of bitvectors. The fact that Z3's implementation of the theory of bitvectors was found to be buggy is particularly surprising due both to its popularity and the fact that it has been fuzzed for more than a year [48] using the fuzzer developed in Brummayer et al. 2009 [17]. For Z3, this seems to be due in part to the fact that we took a particularly aggressive testing approach wherein we strictly required solvers to return satisfiable or unsatisfiable, as opposed to `unknown` [48]; this was significant in one case [154].

A common theme for implementations of the theory of bitvectors was that division and remainder by zero was problematic. SMT-LIB has a well-defined semantics for these

cases, though its correct implementation is challenging. Additionally, there tend to be negative performance consequences of this correct implementation. As a result, all solvers tested are intentionally unsound in their default configurations with this case. Soundness can usually be guaranteed with additional options, though Boolector lacked this option at the time this case study was performed [166]. We believe that these options are rarely enabled, and so we have uncovered a largely untested corner of these solvers.

A related, but more severe, problem exists in the theory of floating point, involving the semantics of `min`/`max` underneath positive and negative zero. In this case, the standard [132] is not entirely clear what the correct behavior is, which required clarification from the standards committee in order to resolve the cited bug [153]. Much like with the theory of bitvectors, MathSAT5 needs an additional option for soundness in this case, which was originally unbeknownst to us. This ultimately led to inputs which had identical, but incorrect, behavior underneath both Z3 and MathSAT5.

5.7 Conclusions

This work has demonstrated that CLP can be applied to the problem of generating SMT formulas with known satisfiability results, and this generation is fast enough to practically test real SMT solvers. A total of 23 bugs in industry-quality SMT solvers were found thanks to this work, of which a remarkable 12 were correctness bugs.

In terms of the overall thesis, this case study helps bolster the argument that CLP is a valid solution to the structured black-box test case generation problem. Generating SMT-LIB formulas with known satisfiability results represents an unparalleled level of structure for a test input, requiring an expressive logic for any hope of accurrate representation. CLP was expressive enough to encode the behavior of SMT-LIB, and it was fast enough to find a number of bugs in popular industrial SMT solvers. That said, this case

study did manage to bring out a weak point in CLP, in that generation was still much slower relative to the other case studies, and it could not practically generate guaranteed unsatisfiable floating point formulas. Considering the complexity of the problem used in this case study, I personally consider such performance degradation acceptable; if CLP had done better, it would supplant literally decades of work on SMT solvers, which seems a farfetched result.

Chapter 6

Case Study: Intelligent Fuzzing of Student Tokenizers and Parsers

6.1 Introduction

This case study looks at how CLP can be utilized by instructors to help gather quantitative and qualitative feedback on how students are performing on assignments. With this application, CLP can assist in semi-automated grading, along with helping to identify when students misunderstand various concepts. Education represents a very different testing domain than that of the other case studies discussed, helping to demonstrate the generality and widespread applicability of CLP to testing. This helps promote the claim that CLP is a general solution to the structured black-box test case generation problem.

This case study looks at one particular student assignment wherein students needed to write a tokenizer, parser, and evaluator for an arithmetic expression language. CLP was used to generate focused tests for each of these components by encoding a *reference solution* directly in CLP itself, and then executing that reference solution "backwards" to produce valid test inputs. This encoding of the reference solution was only possible

thanks to the expressiveness of CLP. Additionally, CLP was able to produce millions of tests within a matter of minutes, demonstrating both the performance of CLP and its capability to generate many tests without difficulty. This further backs my overall thesis.

While there is a significant amount of related work which applies automated testing to the education domain (e.g., [167, 168, 169, 170]), much of this is limited to one specific kind of assignment, and many tools do not produce a deterministic set of test cases. In contrast, CLP can deliver a fixed suite of tests, and it is applicable to more than just a single kind of assignment (further demonstrated in Chapter 7).

As an aside, this chapter is based on work we published in ITiCSE'17 (Citation: [73]; DOI: 10.1145/3059009.3059051; © 2017 ACM).

6.2 Testing Tokenizers, Parsers, and Arithmetic Evaluators With CLP

This section discusses how we use CLP for generating test suites for language tokenizers, parsers, and evaluators. This discussion is separated into two parts:

1. A discussion of how *valid* inputs can be generated (Section 6.2.1). The solutions we test should be able to successfully take valid inputs and produce correct token lists or ASTs, depdending on the component being tested (tokenzers or parsers, respectively).

2. A discussion of how *invalid* inputs can be generated (Section 6.2.2). The solutions we test should reject such inputs, and produce well-defined error messages under such inputs.

We begin this discussion with the generation of valid inputs.

6.2.1 Generating Valid Inputs

Throughout this subsection, we use a running example based on the grammar shown in Figure 6.1, with the corresponding tokens **0** (zero), **1** (one), **-** (minus), **(** (left parentheses) and **)** (right parentheses). While this grammar is admittedly simple, it serves to illustrate all the applicable core concepts, and it forms a subset of the grammar used in the student assignment (Section 6.3). Of special note is that this grammar describes concrete syntax, in contrast to the abstract syntax descriptions used in prior case studies. This use of concrete syntax is unique to this chapter, and it reflects the fact that this chapter focuses on the testing of parsing-related components.

$$
\begin{aligned}
e \in \mathit{Expression} &::= ae \\
ae \in \mathit{AdditiveExppression} &::= pe \mid pe \texttt{ - } ae \\
pe \in \mathit{PrimitiveExpression} &::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{(}e\mathbf{)} \mid \texttt{-}pe
\end{aligned}
$$

Figure 6.1: Small grammar used for running CLP example. Portions shown in **bold** represent tokens.

An executable CLP-based tokenizer applicable to tokenizing the grammar in Figure 6.1 is presented in Figure 6.2, along with a query which will generate valid inputs and corresponding outputs for the tokenizer. The `charToToken` helper procedure maps characters to their token representations. The `tokenize` procedure in Figure 6.2 consists of two rules. The first rule (line 9) states that if there are no characters to tokenize, then there are no tokens produced. The second rule (lines 10-13) states that if the character input begins with a single character, then the tokens produced begin with a single token, where the token is derived from the `charToToken` helper procedure. Furthermore, the second rule recursively calls `tokenize` to process the rest of the input. While this illustrative example has been kept as simple as possible, it is straightforward to add CLP code to allow for whitespace, integers with multiple digits, and multiple-character tokens,

all of which appear in the actual assignment used in our evaluation.

```
1  % charToToken: Character, Token
2  charToToken('0', token_zero).
3  charToToken('1', token_one).
4  charToToken('-', token_minus).
5  charToToken('(', token_lparen).
6  charToToken(')', token_rparen).
7
8  % tokenize: Characters, Tokens
9  tokenize([], []).
10 tokenize([SingleChar|Chars], [SingleToken|Tokens]) :-
11   charToToken(SingleChar, SingleToken),
12   tokenize(Chars, Tokens).
13
14 ?- length(Characters, 5),
15    tokenize(Characters, Tokens).
```

Figure 6.2: CLP-based tokenizer for the language defined in Figure 6.1. `tokenize` takes a list of characters to tokenize along with the tokens produced from the characters, respectively. Line 14 gives a query which will generate all character lists (held in the variable `Characrters`) of length `5` which can be correctly tokenized, returning the corresponding tokens in the variable `Tokens`.

The parser for these tokens can similarly be implemented in CLP, using a standard recursive-descent style. Such a parser is shown in Figure 6.3, along with a query which generates all valid ASTs corresponding to some input tokens.

Unsurprisingly, an evaluator of ASTs can be similarly defined, as is shown in Figure 6.4. In contrast to the tokenizer and the parser, the evaluator code in Figure 6.4 cannot be immediately used as a generator. In particular, there are two problems with this code:

1. We must bound the size of the AST produced in order to use `evaluate` as a generator. Such bounding was easily performed in the tokenizer and parser with the help of `length`, but `length` is not directly applicable to ASTs. This problem, along with solutions, is discussed further in Chapter 7, particularly Section 7.3.

```
1  % parseExpression: Tokens, Exp
2  parseExpression(Tokens, Exp) :-
3    parseExpression(Tokens, [], Exp).
4
5  % parseExpression: InputTokens, OutputTokens, Exp
6  parseExpression(Input, Output, Exp) :-
7    parseAdditiveExpression(Input, Output, Exp).
8
9  % parseAdditiveExpression: InputTokens, OutputTokens, Exp
10 parseAdditiveExpression(Input, Output, Exp) :-
11   parsePrimaryExpression(Input, Output, Exp).
12 parseAdditiveExpression(
13     Input1, Output, exp_subtract(Left, Right)) :-
14   parsePrimaryExpression(Input1, [token_minus|Input2], Left),
15   parseAdditiveExpression(Input2, Output, Right).
16
17 % parsePrimaryExpression: InputTokens, OutputTokens, Exp
18 parsePrimaryExpression([token_zero|Rest], Rest, exp_zero).
19 parsePrimaryExpression([token_one|Rest], Rest, exp_one).
20 parsePrimaryExpression([token_lparen|Input], Output, Exp) :-
21   parseExpression(Input, [token_rparen|Output], Exp).
22 parsePrimaryExpression(
23     [token_minus|Input], Output, exp_unary_minus(Exp)) :-
24   parsePrimaryExpression(Input, Output, Exp).
25
26 ?- length(Tokens, 3),
27    parseExpression(Tokens, Exp).
```

Figure 6.3: CLP-based parser for the language defined in Figure 6.1. `parseExpression` of arity 3 (lines 6-7), `parseAdditiveExpression`, and `parsePrimaryExpression` all take an input list of tokens, an output list of remaining tokens, and the expression produced from the input list of tokens. `parseExpression` of arity 2 (lines 2-3) calls `parseExpression` of arity 3 (line 3), and stipulates that there must be no tokens remaining (i.e., the tokens remaining is an empty list, `[]`). Line 26 gives a query which will generate all expressions (held in variable `Exp`) which can be represented with `3` tokens, along with the actual input tokens (held in variable `Tokens`).

2. The **is** operator (used on lines 7 and 10 of Figure 6.4) requires that the input expression (on the righthand side) be fully instantiated. This is only true if the input expression (the first parameter to `evaluate`) is entirely known (i.e., ground), which

```
1  % evaluate: Exp, Int
2  evaluate(exp_zero, 0).
3  evaluate(exp_one, 1).
4  evaluate(exp_subtract(LeftExp, RightExp), Result) :-
5    evaluate(LeftExp, LeftInt),
6    evaluate(RightExp, RightInt),
7    Result is LeftInt - RightInt.
8  evaluate(exp_unary_minus(Exp), Result) :-
9    evaluate(Exp, ExpInt),
10   Result is -ExpInt.
```

Figure 6.4: CLP-based evaluator for the language defined in Figure 6.1. `evaluate` takes an input expression as its first parameter, and returns the integer result of the expression in the second parameter.

is **not** true if `evaluate` is used as an expression generator. Fixing this demands that arithmetic constraint solvers be used (discussed in Chapter 2, Section 2.2.4, as well as Chapter 3, Section 3.3.5).

Rather than attack the aforementioned problems directly, we exploit the fact that the parser already produces ASTs when it is used as a generator. Given that these ASTs are fully known (i.e., they are ground), they serve as suitable inputs for the `evaluate` procedure defined in Figure 6.4. With this in mind, `evaluate` is used to determine the expected evaluation result of an AST, though `parseExpression` (defined in Figure 6.3) is used to actually produce the ASTs.

6.2.2 Generating Invalid Inputs

The generation of invalid inputs poses a challenge. One possibility is to selectively negate premises to produce invalid inputs by construction, as was done for the generation of *almost* well-typed programs in Chapter 4. However, this approach was deemed to be overly complex given the task at hand. As such, we instead employed *mutation-based fuzzing*, as used in LangFuzz [13]. The basic idea with mutation-based fuzzing is to first

generate valid inputs (as with the technique shown in Section 6.2.1), and then selectively *mutate* them by introducing arbitrary edits. These edits can yield invalid inputs which are still intuitively *close* to being valid, as only the partcular edits are invalid. Details regarding exactly what these edits look like for this test suite follow.

For generating invalid inputs for the tokenizer, we insert characters which will never yield valid tokens into an otherwise tokenizable stream of characters. Specifically, we insert `$`, `=` (ensuring it does not follow either `>` or `<`), `=>`, and `=<`. The character `$` was chosen arbitrarily as a representative of an unconditionally invalid character, and the rest of the characters were chosen as they intuitively seem more likely to trigger faults in a buggy tokenizer. For generating invalid inputs for the parser, we first produce a valid list of tokens which can be parsed to form a valid expression. We then insert an arbitrary valid token into the list, either an integer `0` or `2` (arbitrarily chosen), or any other one of a finite list of remaining valid tokens. Because this process may still yield a parsable list of tokens (as when negating a subexpression), we run the CLP-based parser on the newly generated input to ensure that it **fails**, thus ensuring the input is invalid. While these approaches to generating invalid inputs for the tokenizer and parser are simplistic, we have nonetheless found them to be effective at finding faults in student code.

As for the evaluator, relatively few ASTs act as invalid inputs. By construction, the AST definition in both the Java and CLP reference solutions does not allow for the construction of ASTs with nonsensical structure. With this in mind, the only significant edge case which can be safely deemed "invalid" is that of cases which trigger division by zero, which is supposed to be specially handled by student solutions. We observed that a significant number of the generated valid parser outputs (ASTs produced as described in Section 6.2.1) would attempt to perform division by zero without any outside intervention. As such, we re-used these ASTs as inputs to the evaluator, along with a record of what the AST should evaluate to (be it a number or a trigger for division by zero).

6.3 Student Programming Assignment

The programming assignment used as a case study was given in a second-year programming course in advanced application programming. Students were given code for a working interpreter of infix arithmetic expressions, with separate Java classes for:

- A finite state automaton based tokenizer (along with classes for various kinds of tokens)
- A recursive descent parser that corresponded exactly to the grammar given which produced an AST (along with classes for various AST nodes)
- An interpreter based on a pre-order traversal of the AST

To simplify the assignment, it was assumed that all constants and expressions would evaluate to integers. The given code was capable of interpreting expressions involving addition (+), subtraction (-), multiplication (*), integer division (/), unary minus (-), and parentheses (()). The students were also given a grammar in EBNF for the language supported by the interpreter, shown in Figure 6.5. The students were then required to add support for six relational operators, (<, <=, >, >=, ==, !=) (each of which returns either `0` or `1` based on the truth value of the comparison), as well as exponentiation (**), as reflected in the modified grammar of Figure 6.6. Note that both grammars operate over concrete syntax as opposed to abstract syntax, reflecting the fact that the assignment requires writing an operational parser.

The intent of requiring students to add this particular set of new features was that they necessitated the students to be able to handle several cases which are not in the given code:

- Tokens involving multiple characters

$$i \in \mathbb{Z}$$

$$\begin{aligned}
e \in \textit{Expression} &::= ae \\
ao \in \textit{AdditiveOperation} &::= \texttt{+} \mid \texttt{-} \\
ae \in \textit{AdditiveExpression} &::= me\ (ao\ me)^* \\
mo \in \textit{MultiplicativeOperation} &::= \texttt{*} \mid \texttt{/} \\
me \in \textit{MultiplicativeExpression} &::= pe\ (mo\ pe)^* \\
pe \in \textit{PrimaryExpression} &::= \texttt{(}\ e\ \texttt{)} \mid i \mid \texttt{-}\ pe
\end{aligned}$$

Figure 6.5: EBNF grammar defining the language the given parser accepts.

$$i \in \mathbb{Z}$$

$$\begin{aligned}
e \in \textit{Expression} &::= ce \\
co \in \textit{ComparisonOperation} &::= \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \\
ce \in \textit{ComparisonExpression} &::= ae\ (co\ ae)^* \\
ao \in \textit{AdditiveOperation} &::= \texttt{+} \mid \texttt{-} \\
ae \in \textit{AdditiveExpression} &::= me\ (ao\ me)^* \\
mo \in \textit{MultiplicativeOperation} &::= \texttt{*} \mid \texttt{/} \\
me \in \textit{MultiplicativeExpression} &::= ee\ (mo\ ee)^* \\
ee \in \textit{ExponentiationExpression} &::= pe\ \texttt{**}\ ee \mid pe \\
pe \in \textit{PrimaryExpression} &::= \texttt{(}\ e\ \texttt{)} \mid i \mid \texttt{-}\ pe
\end{aligned}$$

Figure 6.6: EBNF grammar defining the language students must define a parser for. This is very similar to the grammar defined in Figure 6.5, except for the exponentiation (**) operation and the new *ComparisonOperation* and *ComparisonExpression* productions.

- Tokens which share common prefixes. For example, < is a prefix of <=, and both are individually valid tokens.

- Right-associative operators, specifically exponentiation (**). The given code contains only left-associative operators.

6.4 Evaluation

During Fall 2016, students submitted solutions which were then automatically graded via a traditional handcrafted test suite composed of 230 tests. The assignment had an option for either individual or pair submission. The study is based on 48 submissions where either the sole author or both partners gave informed consent.

We then used the CLP-based technique described in Section 6.2 to generate a test suite composed of 7,291,812 tests, specifically tests focused on the tokenizer, parser, and evaluator for the grammar from Figure 6.6. Where possible, the same test input was reused to test multiple components. For example, consider the following test input:

```
1 - 1
```

This input should tokenize, parse, and evaluate successfully down to the value `0`. As such, it can be used as a test each component individually; that is, the characters serve as a tokenizer test, the tokens corresponding to it serve as a parser test, and the expression produced by the parser serves as an evaluator test. If a solution failed part of the tokenize, parse, and evaluate chain, a correct intermediate form was substituted so the remaining components could be individually tested. For example, if the student's tokenizer failed to tokenize the above input, we would mark it as a tokenizer failure, but would nonetheless then try to test the student's parser with the correct tokens corresponding to the above input.

We first ran both test suites on both the Java and CLP reference solutions, which were coded by two separate individuals in order to reduce the chance that they suffered from common bugs. Both solutions passed all tests, strongly suggesting that both solutions were correct.

We then ran both the handcrafted and the CLP-generated tests against the student solutions. Our raw data, therefore, consisted of results for 230 handcrafted tests and

approximately seven million CLP-generated tests for each of the 48 student solutions. Looking at the data, the following conclusions were immediately reached:

- There was no case where a solution passed all of the CLP-based tests, but failed at least one of the handcrafted tests. This indicates that the CLP-based test suite was at least as powerful as the handcrafted test suite.
- Of the 40 solutions that passed all of the handcrafted tests, only 30 of those passed all of the CLP-based tests. This indicates that there are cases where the CLP-based test suite was strictly more powerful than the handcrafted test suite.

Eight of the solutions failed tests on both the handcrafted test suite and the CLP-based test suite. Because failures occurred on both test suites for these solutions, a more sophisticated approach was necessary in order to draw any meaningful conclusions for these cases.

6.4.1 Test Suite Comparison via Equivalence Classes

We observe that solutions can be separated into equivalence classes based on the tests they fail. That is, it is common for multiple solutions to fail the exact same set of tests, suggesting that different solutions share the same underlying defects.

We first partitioned the 48 solutions on the basis of which tests were failed on the handcrafted tests, which yielded only six equivalence classes. These six classes are shown visually in the top half of Figure 6.7. They are as follows:

- Three singleton classes.
- One class of two solutions.
- One class of three solutions.

- One class of 40 solutions. These are the solutions that failed none of the hand-crafted tests.

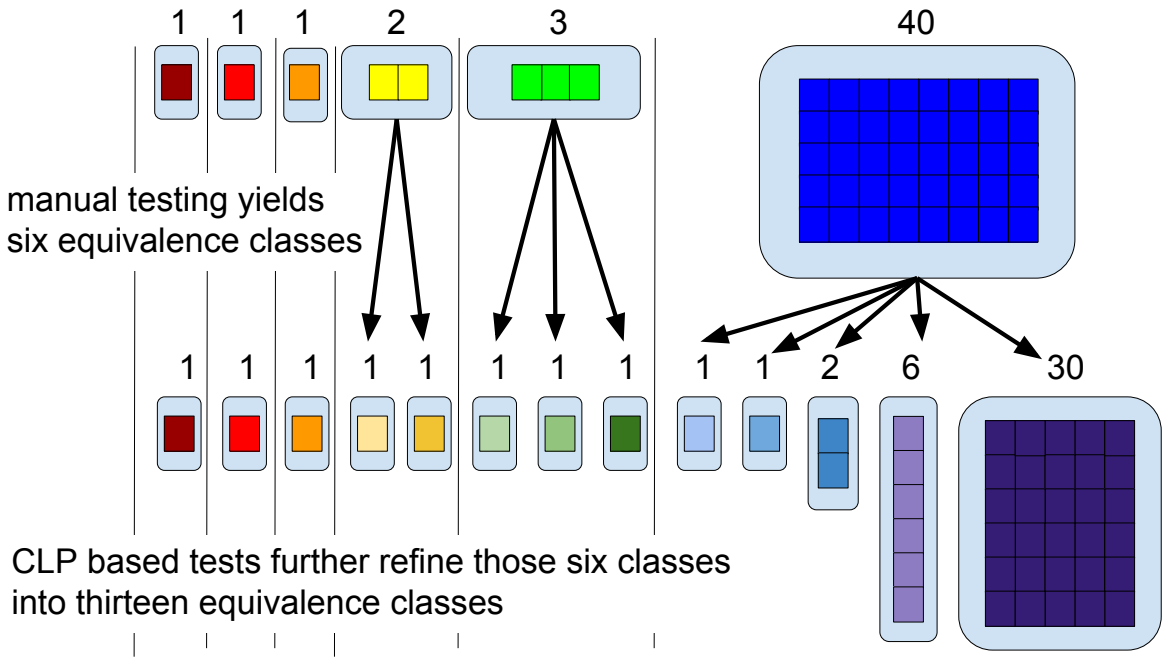

Figure 6.7: Equivalence classes of student submissions, based on which tests they failed.

While the small number of equivalence classes may raise the question of plagiarism, follow-up with Moss [171, 172] showed that plagiarism alone cannot account for this effect.

Partitioning the 48 solutions based on which tests were failed by the CLP-based test suite resulted in a a further refinement of these six equivalence classes into thirteen classes, as shown in the bottom half of Figure 6.7. The partitions produced via the CLP-based test suite are represented visually by the arrows in Figure 6.7. Each division represents a case where the CLP-based tests are revealing code behavior that the manual test suite missed.

The set of solutions that passes all of the CLP tests contains only 30 solutions; ten of the solutions from the original equivalence class of 40 failed some of the CLP-based

tests, resulting in four additional equivalence classes. The original classes of two and three were also further split into singletons. Each of these splits represents a case where the CLP-based tests were able to distinguish among solutions with finer granularity, thus potentially revealing additional defects missed by the manual test suite.

6.4.2 Code Inspection

We did a manual code inspection of representative solutions from both sides of each equivalence class split to qualitatively learn more about what these splits signified. It is tempting to assume that each such split indicates a new bug or set of bugs. Our explorations show that this is often, though not always the case. The other possibility is that the solutions have made similar errors but errors that differ enough in the particular way they are incorrect, such that they pass different numbers of tests. However, each division did offer some insight into ways in which students approached the problem, and the types of mistakes they made in their code.

The original large equivalence class of 40 students is broken up five ways, overall revealing that 10 students who passed all the handcrafted tests nonetheless still had bugs in their code (an indication that if the test suite had been more powerful, there may have been more rigor and fairness in the grades assigned). These 10 were divided into two singletons, a pair, and a group of 6. One of the singletons was a solution that, although it was correct from the standpoint of an end user, failed many parser tests because the testing code relied on the `.equals()` method of one of the student-defined AST classes to work properly (when comparing actual vs. expected results), however the student failed to override this method. This was a case where the handcrafted test suite was deficient. The three remaining classes were all characterized by various errors involving negative exponents, bringing to light the fact that the instructors had completely overlooked

testing for the cases of zero and negative exponents (focusing the manual tests instead on the right-associativity of the operator).

Students that correctly handled the exponent operator used a variety of approaches. Some students computed a value using Java's `Math.pow()` method and then cast the result to an integer value. Others used loops which performed repeated multiplication for non-negative exponents, and repeated division for negative exponents. With these loops, some handled x^0 as a special case while others initialized a product value to `1`, and then used a loop to repeatedly multiply by the base of the exponent (so that zero iterations of the loop naturally returns the correct value).

Figures 6.8, 6.9, and 6.10 show three incorrect approaches to computing exponents with loops all taken from the group of 40, each passing a different number of CLP-based tests. Each is shown with only minor editing related to spacing and variable names. Each is intented to compute the result of `left ** right`, ultimately returning this value. Figure 6.8 correctly computes positive and zero exponents, and has an incorrect attempt at negative exponents. Figure 6.9 correctly calculates positive and zero exponents, but has no code for negative exponents. Figure 6.10 calculates only positive exponents correctly.

There were two other equivalence classes that were further partitioned by the CLP-based testing: an equivalence class of two, and one of three, each of which was refined into singletons by the CLP-based testing. Code inspection of the class of two solutions revealed that both of the solutions failed the hand-written tests for both the new comparison operators and the exponentiation operator. What distinguished one solution was an error in the finite state automaton; it failed to recognize that the ***** token was a prefix of the ****** token, and did not set up a state transition from the state for the multiplication operator to the state for the exponentiation operator.

The equivalence class of three solutions was refined further by CLP-generated tests into three singletons. All three solutions shared a common bug related to an improperly

```
1  int result = left;
2  if (right == 0) {
3    return 1;
4  } else if (right < 0) {
5    for (int i = 0; i < (right * -1) - 1; i++) {
6      result = result / left;
7    }
8    return result;
9  } else {
10   for (int i = 0; i < right - 1; i++) {
11     result = result * left;
12   }
13   return result;
14 }
```

Figure 6.8: Attempts to handle negative exponents, but incorrectly.

```
1 int result = 1;
2 for (int i = 0; i < right; i++) {
3   result *= left;
4 }
5 return result;
```

Figure 6.9: Implicitly assumes that the exponent will be non-negative.

```
1 int result = left;
2 for(int i = 1; i < right; i++) {
3   result = result * left;
4 }
5 return result;
```

Figure 6.10: Implicitly assumes that the exponent will be positive.

structured `if`/`else`. The first of these had a bug not found in the other two, related to an issue of `==` vs. `.equals()` for objects. All three had problems related to calculation of exponents, but the third solution handled an exponent of `0` correctly, in contrast to the first two solutions.

Our overall conclusion from these qualitative observations is that while each division

Test Suite	Line	Method	Branch
Handcrafted	77%	74%	62%
CLP-based	80%	83%	62%
CLP Improvement (Percentage Points)	3	9	0

Table 6.1: Average code coverage metrics for the two test suites across all student solutions.

between equivalence classes is an interesting place to look for defects, it is not necessarily the case that each corresponds to a particular discrete bug. This sort of difficulty in defining exactly what "bug" means is a known problem in Software Engineering research (e.g., [173]), and this work makes this apparent in the context of educational assessment.

6.4.3 Test Suite Code Coverage

While the CLP-based test suite exposed clear deficiencies in the handcrafted test suite, we wondered as to whether or not these deficiencies could have been discovered ahead of time with a more traditional approach. To this end, we measured average code coverage across all students for the two test suites, the results of which are shown in Table 6.1. As shown in Table 6.1, there are uniformly relatively low values for all coverage metrics employed. This can be explained by the presence of debugging-oriented code in student solutions, as well as methods which are good practice to implement but not directly under test (e.g., `hashCode()`, `toString()`, and `equals()`). It is difficult to fairly prune out such code, because some students did nonetheless use it during testing. With this in mind, most student solutions contained a significant amount of dead code independent of the test suite used, explaining the low coverage metrics.

While the CLP-based test suite tended to get better code coverage than the hand-crafted test suite, the improvements are often marginal at best. This leads us to conclude that code coverage can be a misleading measure of a test suite's effectiveness. This motivates the direct measurement of the defect-finding power of a test suite, as can be done with CLP.

6.5 Conclusions

This case study has shown that CLP can be applied to the educational domain, where it was successful in finding a number of bugs missed by traditional testing practices. This serves to show not only the bug-finding effectiveness of CLP, but also that CLP is a general solution; the educational domain is quite different than that of the sort of industrial case studies previously discussed, but CLP is still nonetheless applicable. Moreover, CLP was able to generate millions of tests within a matter of minutes, and these tests required CLP to reason about the behaviors of tokenizers, parsers, and evaluators. This goes to show that CLP is not only fast, but that it can reason about relatively complex parsing-based ideas. Overall, this case study further strengthens my thesis that CLP is an effective solution to the generalized structured black-box test case generation problem.

Chapter 7

Case Study: Generating Polymorphic Programs for Testing Student Typecheckers

7.1 Introduction

Generating full well-typed programs is challenging. While the prior work on Rust (see Chapter 4) demonstrated how to generate diverse well-typed programs, there was a catch: no single well-typed program generator understood the entire type system. The end result is that the programs generated were fragmented, as each well-typed program generator could only generate within the fragment of the type system it understood.

In this case study, I seek to reason about an **entire** complex type system with a **single** CLP-based test case generator. Specifically, I want to generate well-typed SimpleScala programs, where SimpleScala is an educational language I developed for UCSB's Programming Language's course (CS162). SimpleScala features abstract data types, parametric polymorphism, generics, and higher-order functions, of which all introduce their

own generation challenges. Some of these challenges were discussed in Chapter 4, though these challenges were significantly easier to handle in that chapter given the fragmented nature of the generators employed.

Generating well-typed SimpleScala programs serves as an excellent case study into the power of CLP. As with Rust (chapter 4), this pushes the expressibility limit beyond anything ever attempted in outside work. However, this case study is specific to the educational domain, as with Chapter 6. As such, this serves to bolster the argument that CLP is broadly applicable. Moreoever, CLP was able to generate a sizable test suite containing over 100K inputs within several hours, demonstrating that the performance of CLP is more than adequate for this application. Overall, this case study serves as further evidence that CLP is a good solution to the generalized structured black-box test case generation problem.

As an aside, while this work is not published, we are in discussions to do a similar study as performed in Chapter 6. While I have used this work in CS162 with great success, I cannot discuss the specifics as I did not submit a relevant request to the institutional review board.

7.2 SimpleScala Language

This section describes the SimpleScala language. Ultimately I want to generate well-typed programs in this language with CLP.

7.2.1 General Design and Syntax

SimpleScala was designed with the intention of giving the same look and feel of Scala, at least from the perspective of a student who is first learning Scala and functional programming. The abstract syntax of SimpleScala is presented in Figure 7.1. This

language contains the following notable features:

- Relatable base types, namely `String` (**string** in the formalism), `Boolean` (**boolean** in the formalism), `Int` (**integer** in the formalism), and `Unit` (**unitType** in the formalism).
- Named functions, introduced with the **def** keyword.
- Variables introduced with the **val** keyword.
- Conditionals, introduced with the **if** and **else** keywords.
- Tuples, introduced with comma-separated expressions in parentheses. These can be accessed using Scala's `._n` notation, for some positive number `n`. Notably, tuple access is a builtin operation, as opposed to a method on a tuple object (SimpleScala notably lacks objects in any object-oriented sense).
- Higher-order functions, introduced with `=>`. Unlike Scala, SimpleScala higher-order functions require that the function parameter is always annotated (i.e., there is no type inference). Additionally, SimpleScala higher-order functions always take *exactly one* parameter; the **unitType** type and **unit** value are provided as dummy types and values for functions which wish to take no values, and tuples / currying are used for functions which wish to take more than one value.
- Algebraic data types, using Scala's notation to create instances of these types and pattern match on them (using the **match** keyword). While these look and feel very similar to Scala's approach of using `case class`es for similar purposes, the underlying theory and implementation is radically different. As such, they are defined with the **algebraic** keyword, which serves as an indicator to students that

these behave in subtly, but fundamentally, different ways than do Scala's `case class`es.

- The ability to define parametric polymorphism (type variables on code) using square brackets on a **def**. Unlike with Scala, even if a computation does not use type variables, empty square brackets are required.

- The ability to define generics (type variables on data) using square brackets in an **algebraic** definition. Even if no type variables are used in a given **algebraic** definition, empty square brackets must be provided, in contrast to non-generic `case class` definitions in Scala.

- Programs (*prog*) consist of a series of user-defined type definitions ($\overrightarrow{tdefs}$), followed by a series of user-defined functions ($\overrightarrow{def}$), followed by a single expression (e) which serves as a program entry point.

7.2.2 Type Domains

The typing rules for this language utilize a number of formal definitions which are provided in Figure 7.2. A brief description of each one of these definitions follows:

- *fdefs*: Records named functions defined with the **def** keyword in a convenient format. The parser ensures that this mapping is unique (i.e., no two functions share the same name). Specifically, *fdefs* maps function names to to 3-tuples of:

  1. The type variables introduced (i.e., those in scope) for the function
  2. The input type of the function
  3. The output type of the function

$$x \in \mathit{Variable} \qquad \mathit{str} \in \mathit{String} \qquad b \in \mathit{Boolean} \qquad i \in \mathbb{Z} \qquad n \in \mathbb{N}$$
$$\mathit{fn} \in \mathit{FunctionName} \qquad \mathit{cn} \in \mathit{ConstructorName} \qquad \mathit{un} \in \mathit{UserDefinedTypeName}$$
$$T \in \mathit{TypeVariable}$$

$$\begin{aligned}
\tau \in \mathit{Type} &::= \textsf{string} \mid \textsf{boolean} \mid \textsf{integer} \mid \textsf{unitType} \\
&\mid \tau_1 \to \tau_2 \mid (\ \vec{\tau}\ ) \mid \mathit{un}[\vec{\tau}] \mid T \\
e \in \mathit{Exp} &::= x \mid \mathit{str} \mid b \mid i \mid \textsf{unit} \mid e_1 \oplus e_2 \\
&\mid (x : \tau) \Rightarrow e \mid e_1(e_2) \mid \mathit{fn}[\vec{\tau}](e) \\
&\mid \textsf{if}\ (e_1)\ e_2\ \textsf{else}\ e_3 \\
&\mid \{\overrightarrow{\mathit{val}}\ e\} \\
&\mid (\ \vec{e}\ ) \mid e._n \\
&\mid \mathit{cn}[\vec{\tau}](e) \mid e\ \textsf{match}\ \{\overrightarrow{\mathit{case}}\} \\
\mathit{val} \in \mathit{Val} &::= \textsf{val}\ x = e \\
\mathit{case} \in \mathit{Case} &::= \textsf{case}\ \mathit{cn}(x) \Rightarrow e \mid \textsf{case}\ (\ \vec{x}\ ) \Rightarrow e \\
\oplus \in \mathit{Binop} &::= + \mid - \mid \times \mid \div \mid \wedge \mid \vee \mid < \mid \leq \\
\mathit{tdef} \in \mathit{UserDefinedTypeDef} &::= \textsf{algebraic}\ \mathit{un}[\overrightarrow{T}] = \overrightarrow{\mathit{cdef}} \\
\mathit{cdef} \in \mathit{ConstructorDefinition} &::= \mathit{cn}(\tau) \\
\mathit{def} \in \mathit{Def} &::= \textsf{def}\ \mathit{fn}[\overrightarrow{T}](x : \tau_1) : \tau_2 = e \\
\mathit{prog} \in \mathit{Program} &::= \overrightarrow{\mathit{tdef}}\ \overrightarrow{\mathit{def}}\ e
\end{aligned}$$

Figure 7.1: SimpleScala syntax.

Because the functions defined do not change throughout typechecking or program execution, *fdefs* is treated as a global constant.

- *tdefs*: Records user-defined types defined with the **algebraic** keyword in a convenient format. The parser ensures that this mapping is unique (i.e., no two user-defined types share the same name). Specifically, *tdefs* maps user-defined type names to pairs of:

  1. The type variables introduced (i.e., those in scope) for the data type definition

2. A mapping of each constructor of the user-defined type to the type each constructor expects

Because the user-defined type definitions do not change throughout typechecking or program execution, *tdefs* is treated as a global constant.

- *cdefs*: Records a backwards mapping of constructor names to the name of the user-defined type the constructor creates. The parser ensures that this mapping is unique (i.e., each constructor name is unique, even across types). For the same reasoning as with *tdefs*, *cdefs* is treated as a global constant.

- *tscope*: Records the type variables which are currently in scope. If we are currently typechecking the program's entry point (i.e., e in *prog*), then $tscope = \emptyset$. If we are currently typechecking the expression inside a function defined with **def** with name *fn*, then $tscope = \texttt{first}(fdefs(fn))$, where `first` gets the first element in a tuple. The program entry point and each **def** can be typechecked independently of each other, and *tscope* will never change throughout typechecking the individual unit (i.e., the program entry point or a **def**). As such, *tscope* is treated as a global variable.

- Γ: Maps the variables in scope along with their recorded types. Because variables can be added in scope at a number of points, Γ must be threaded through the typing rules. Additionally, because SimpleScala follows lexical scoping, Γ only needs to be passed down, not up. That is, with the exception of shadowing, it is not possible for a variable introduced in one scope to influence a variable in another scope, so there is no need to return which variables were in scope for a given expression. This is considered standard.

$$
\begin{aligned}
fdefs \in NamedFunctionDefs &= FunctionName \rightarrow (\overrightarrow{TypeVariable} \times Type \times Type) \\
tdefs \in TypeDefs &= UserDefinedTypeName \rightarrow \\
&\qquad (\overrightarrow{TypeVariable} \times (ConstructorName \rightarrow Type)) \\
cdefs \in ConstructorDefs &= ConstructorName \rightarrow UserDefinedTypeName \\
tscope \in TypeVarsInScope &= \overline{TypeVariable} \\
\Gamma \in TypeEnv &= Variable \rightarrow Type
\end{aligned}
$$

Figure 7.2: Various definitions used in the typing rules.

7.2.3 Typing Rules and Helper Functions

The actual typing rules are shown in Figures 7.3 and 7.4, with Figure 7.3 showing all rules except for pattern matching, and Figure 7.4 showing only the rules for pattern matching. The rules have been split up only for reasons of space.

A variety of helper functions are employed in these rules. Most of these functions are intuitive and relatively standard, so their full definitions have been provided in Appendix E as opposed to being directly in this chapter. A brief description of each helper function used is provided below for convenience:

- `keys`: Returns a set of keys in the given map.
- `typeOk`: Returns **true** if all type variables used in the given type are in scope. If no type variables are used in the given type, it simply returns **true**.
- `typeOkList`: Like `typeOk`, but it operates over a list of types as opposed to just a single type.
- `typeReplace`: This is responsible for replacing type variables with actual types at the appropriate times (i.e., when a named function is called, or when an instance of a user-defined type is created). For example, consider the call

`typeReplace`$([A], [\textsf{integer}], (\textsf{boolean}, A))$, where the notation $[A]$ indicates a list holding type variable A, and so on. This will replace each instance of type variable A in type $(\textsf{boolean}, A)$ with **integer**, resulting in the new type $(\textsf{boolean}, \textsf{integer})$.

- `blockEnv`: Produces a new type environment where the given list of variable/-value definitions (defined with the **val** keyword) are added to the provided type environment.

- `tupleTypes`: Given a list of expressions and a type environment, returns a list of types, where each type in the returned list corresponds to an expression in the input list at the same position of the list. For example, if given $[1, \textsf{true}, \textsf{unit}]$ as expressions under any type environment, this would return $[\textsf{integer}, \textsf{boolean}, \textsf{unitType}]$.

- `tupleAccess`: Given a list of types and a positive number n, yields the nth element of the list in a 1-indexed manner. If there is no such element in the input list (e.g., when only two types are provided, but $n = 5$), this function cannot be applied, and attempting to apply it will trigger handling for ill-typedness.

- `tupGamma`: Produces a new type environment where the given list of variables are in scope, each associated with a type from the given list of types. For example, consider the call `tupGamma`$([x, y], [\textsf{boolean}, \textsf{integer}], \Gamma)$. This will produce a new type environment wherein x maps to **boolean**, and y maps to **integer**. The name of the function reflects the fact that this is used only for pattern matching on tuples (hence the "`tup`" part).

- `casesOk`: Specific to pattern matching involving user-defined types, this makes sure that there is exactly one case for each possible constructor of a type, and that all possible constructors are accounted for. If a case has been duplicated, is missing, or

does not match up with the appropriate user-defined type in play, then this returns **false**. Otherwise, `casesOk` returns **true**.

- `casesTypes`: Determines the type of the branch of each case for pattern matching on user-defined types, yielding a list of types, one for each case. This needs the actual cases, the current type environment, the generic type variables in scope for whatever user-defined type is in play, the types we want to replace the type variables with, and a mapping of constructor names to the types that each one of the constructors expects.

- `asSingleton`: Given a list of types, gets the first element of the list, but only if each element of the list is identical. If there are two non-identical elements in the given list, `asSingleton` cannot be applied. In context, `asSingleton` is used to ensure that the body of each branch in a pattern match on a user-defined type returns the same type. If two branches differ in type, then `asSingleton` triggers handling for ill-typedness.

Overall, the typing rules should be straightforward, albeit dense.

7.3 A Naive CLP-Based Generator for Well-Typed SimpleScala Programs

This section introduces a core fragment of a CLP-based test case generator for well-typed programs in SimpleScala. This section also explains the sort of problems that arise with such a naive generator.

The typing rules presented in Figures 7.3 and 7.4 can be used to implement a type-checker for SimpleScala in any arbitrary language. For example, in CS162, students

$$\frac{x \in \texttt{keys}(\Gamma) \quad \tau = \Gamma(x)}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash \mathit{str} : \textsf{string}}\ (\textsc{string}) \qquad \frac{}{\Gamma \vdash b : \textsf{boolean}}\ (\textsc{bool})$$

$$\frac{}{\Gamma \vdash \textsf{unit} : \textsf{unitType}}\ (\textsc{unit}) \qquad \frac{\Gamma \vdash e_1 : \textsf{integer} \quad \Gamma \vdash e_2 : \textsf{integer}}{\Gamma \vdash e_1 + e_2 : \textsf{integer}}\ (+_{int})$$

$$\frac{}{\Gamma \vdash i : \textsf{integer}}\ (\mathbb{Z}) \qquad \frac{\Gamma \vdash e_1 : \textsf{string} \quad \Gamma \vdash e_2 : \textsf{string}}{\Gamma \vdash e_1 + e_2 : \textsf{string}}\ (+_{string})$$

$$\frac{\otimes \in \{-, \times, \div\} \quad \Gamma \vdash e_1 : \textsf{integer} \quad \Gamma \vdash e_2 : \textsf{integer}}{\Gamma \vdash e_1 \otimes e_2 : \textsf{integer}}\ (\textsc{arithOp})$$

$$\frac{\otimes \in \{\wedge, \vee\} \quad \Gamma \vdash e_1 : \textsf{boolean} \quad \Gamma \vdash e_2 : \textsf{boolean}}{\Gamma \vdash e_1 \otimes e_2 : \textsf{boolean}}\ (\textsc{boolOp})$$

$$\frac{\otimes \in \{<, \leq\} \quad \Gamma \vdash e_1 : \textsf{integer} \quad \Gamma \vdash e_2 : \textsf{integer}}{\Gamma \vdash e_1 \otimes e_2 : \textsf{boolean}}\ (\textsc{relOp})$$

$$\frac{\texttt{typeOk}(\tau_1) \quad \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash (x : \tau_1) \Rightarrow e : \tau_1 \to \tau_2}\ (\textsc{AFun}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ (\textsc{ACall})$$

$$\frac{\begin{array}{l}\texttt{typeOkList}(\vec{\tau_1}) \quad \mathit{fn} \in \texttt{keys}(\mathit{fdefs}) \quad (\overrightarrow{T} \cdot \tau_2 \cdot \tau_3) = \mathit{fdefs}(\mathit{fn}) \quad |\overrightarrow{T}| = |\vec{\tau_1}| \\ \tau_2' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) \quad \tau_3' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_3) \qquad \Gamma \vdash e : \tau_2'\end{array}}{\Gamma \vdash \mathit{fn}[\vec{\tau_1}](e) : \tau_3'}\ (\textsc{NCall})$$

$$\frac{\Gamma \vdash e_1 : \textsf{boolean} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if}\ (e_1)\ e_2\ \textsf{else}\ e_3 : \tau}\ (\textsc{if}) \qquad \frac{\Gamma' = \texttt{blockEnv}(\overrightarrow{\mathit{val}}, \Gamma) \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \{\overrightarrow{\mathit{val}}\ e\} : \tau}\ (\textsc{val})$$

$$\frac{\vec{\tau} = \texttt{tupleTypes}(\vec{e}, \Gamma)}{\Gamma \vdash (\ \vec{e}\ ) : (\ \vec{\tau}\ )}\ (\textsc{tup}) \qquad \frac{\Gamma \vdash e : (\ \vec{\tau}\ ) \quad \tau' = \texttt{tupleAccess}(\vec{\tau}, n)}{\Gamma \vdash e.\_n : \tau'}\ (\textsc{acc})$$

$$\frac{\begin{array}{l}\texttt{typeOkList}(\vec{\tau_1}) \quad \mathit{cn} \in \texttt{keys}(\mathit{cdefs}) \quad \mathit{un} = \mathit{cdefs}(\mathit{cn}) \quad (\overrightarrow{T} \cdot m) = \mathit{tdefs}(\mathit{un}) \\ |\overrightarrow{T}| = |\vec{\tau_1}| \quad \tau_2 = m(\mathit{cn}) \quad \tau_2' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) \quad \Gamma \vdash e : \tau_2'\end{array}}{\Gamma \vdash \mathit{cn}[\vec{\tau_1}](e) : \mathit{un}[\vec{\tau_1}]}\ (\textsc{cons})$$

Figure 7.3: Typing rules for SimpleScala, without pattern matching. The A in AFun and ACall is short for "Anonymous", and the N in NCall is short for "Named". Rules for pattern matching are shown in Figure 7.4.

$$\frac{\Gamma \vdash e_1 : (\ \vec{\tau_1}\ ) \quad |\vec{\tau_1}| = |\vec{x}| \quad \Gamma' = \texttt{tupGamma}(\vec{x}, \vec{\tau_1}, \Gamma) \quad \Gamma' \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \textbf{match}\ \{(\textbf{case}\ (\ \vec{x}\ ) \Rightarrow e_2) :: []\} : \tau_2}\ (\text{M-TUP})$$

$$\frac{\Gamma \vdash e : un[\vec{\tau_1}] \quad un \in \texttt{keys}(tdefs) \quad \texttt{casesOk}(\overrightarrow{case}, un) \quad (\overrightarrow{T} \cdot m) = tdefs(un) \quad |\overrightarrow{T}| = |\vec{\tau_1}| \quad \vec{\tau_2} = \texttt{casesTypes}(\overrightarrow{case}, \Gamma, \overrightarrow{T}, \vec{\tau_1}, m) \quad \tau_3 = \texttt{asSingleton}(\vec{\tau_2})}{\Gamma \vdash e\ \textbf{match}\ \{\overrightarrow{case}\} : \tau_3}\ (\text{M-CONS})$$

Figure 7.4: Typing rules handling pattern matching in SimpleScala. The typing rules for everything else are shown in Figure 7.3.

must implement a typechecker for SimpleScala, given these formal rules. For our purposes, we can go through the same process, but use CLP as an implementation language instead of Scala. Unsurprisingly, this CLP implementation process immediately yields a typechecker for SimpleScala. However, what is more interesting for our purposes is that this typechecker can be readily adapted to become a *generator* of well-typed programs, similar to the approach of writing a reference solution in Chapter 6.

A core fragment of this CLP-based typechecker implementation is shown in Figure 7.5, which implements a subset of the rules shown in Figure 7.3. To be clear, the purpose of showing this code is not to explain exactly how to implement a typechecker in CLP, but rather to have a starting generator to improve upon. Additionally, there are certain implementation details which have been elided which are not particularly relevant to the discussion (e.g., CLP does not have a real concept of a global variable, so we need to pass "global" information around in addition to everything else).

The code shown in Figure 7.5 behaves correctly when used as a typechecker. When given an expression, it will either produce the type of that expression if the expression is well-typed, or fail if the expression is ill-typed. In an ideal world, we could also perform the *reverse* operation: that of giving a type and then asking for an expression of that type. Or, better yet, give no type and simply ask for an arbitrary well-typed expression of an arbitrary type. However, this code has a number of problems which make it unsuitable

```
1  % TypeEnv: [pair(VariableName, Type)]
2  % typeof: TypeEnv, Exp, Type
3  %
4  % VAR
5  typeof(Gamma, variable(X), Type) :-
6    member(pair(X, Type), Gamma).
7  % STRING
8  typeof(_, strValue(_), stringType).
9  % INTEGER
10 typeof(_, intValue(_), intType).
11 % + (int)
12 typeof(Gamma, binop(E1, plus, E2), intType) :-
13   typeof(Gamma, E1, intType),
14   typeof(Gamma, E2, intType).
15 % + (string)
16 typeof(Gamma, binop(E1, plus, E2), stringType) :-
17   typeof(Gamma, E1, stringType),
18   typeof(Gamma, E2, stringType).
19 % AFUN
20 typeof(Gamma, afun(X, T1, E), funType(T1, T2)) :-
21   typeOk(T1),
22   addMap(Gamma, X, T1, GammaPrime),
23   typeof(GammaPrime, E, T2).
24 % ACALL
25 typeof(Gamma, acall(E1, E2), T2) :-
26   typeof(Gamma, E1, funType(T1, T2)),
27   typeof(Gamma, E2, T1).
```

Figure 7.5: CLP implementation of a typechecker for SimpleScala. The `member` procedure is standard, and is used to select a member of a list. The `addMap` procedure is assumed to have been implemented elsewhere, and it adds a key/value pair to an existing map, yielding a new map. The `typeOk` procedure implements the helper function with the same name in the helper functions defined in Appendix E.

for generation purposes. In particular, there are four major problems, which are fully described and solved in each of the four following subsections. These are summarized below:

1. The search space is infinite (discussed in Section 7.3.1).

2. String and integer values are not bound to actual strings and integers (discussed in Section 7.3.2).

3. Variable names are not bound to actual names (discussed in Section 7.3.3). While this seems like the same problem as string and integer values being unbounded on the surface, this problem causes very different behavior, and the solution employed is significantly different as well. As such, this discussion is separated out from string and integer values.

4. Cyclic terms are currently possible (discussed in Section 7.3.4).

Each subsection starts from Figure 7.5, and shows how to fix the particular problem of focus in isolation. In practice, each fix would need to be applied concurrently. I intentionally do not perform this sort of concurrent fixing in this discussion in order to simplify the presentation.

7.3.1 Infinite Search Space

Probably the most immediate problem is that the search space is infinite when the code in Figure 7.5 is used as a generator. The source of this problem lies in the fact that there are an infinite number of well-typed programs in SimpleScala. While this is not problematic in and of itself, it leads to problems when combined with CLP's depth-first search strategy. Specifically, most rules will never be executed, as CLP will always choose to produce ever more deeply nested expressions. To illustrate, consider the following query, which asks for an arbitrary well-typed expression (`Exp`) of an arbitrary type (`Type`):

```
?- typeof([], Exp, Type).
```

The first four expressions generated by this query are below, one per line (stripping whitespace to avoid line wrapping):

```
1 strValue(_);
2 intValue(_);
3 binop(intValue(_),plus,intValue(_));
4 binop(intValue(_),plus,binop(intValue(_),plus,intValue(_)))
```

It is no accident that the fourth result embeds the third result to the right. This is a direct consequence of the fact that in the code listing in Figure 7.5 lists the rule handling $+_{int}$ (from Figure 7.3) *before* the rule handling $+_{string}$ (also from Figure 7.3). Because CLP uses a depth-first search strategy and tries rules in the same order as they are listed, it will *always* be the case that $+_{int}$ will be tried before $+_{string}$. This, combined with the fact that we currently can always generate a more deeply-nested expression than the expression previously generated, means that we will keep generating ever more deeply-nested uses of the $+_{string}$ rule, and effectively ignore all other rules.

This problem of "spamming" the same rule is undesirable, as it prevents us from generating diverse sets of programs. This problem is fixed rather easily by adding a bound of some sort to restrict the state space. For example, we can add a bound on the depth of expressions generated. If we ever attempt to generate an expression which is deeper than our allotted bound, then failure is triggered, forcing the CLP engine to backtrack to another choice. On backtracking, the engine will select a *different* rule than what was previously selected, leading to the generation of a different program.

To see this bound in action, the code presented in Figure 7.5 has been instrumented with bounds in Figure 7.6. With this updated code, we can issue the following modified query, which enforces that we will never generate a program that is more than one expression (1) deep:

```
?- typeof(1, [], Exp, Type).
```

The first four results of this modified query are below:

```
1 strValue(_);
2 intValue(_);
3 binop(intValue(_), plus, intValue(_));
4 binop(strValue(_), plus, strValue(_))
```

In contrast to the previous query, the current query yields something different than a more deeply-nested `binop` with the fourth result. Instead, the current query ends up producing `binop` with strings (`strValue`) as opposed to integers (`intValue`), which is an expression which is unique to the current query. As such, this ultimately solves the problem of having an infinite search space.

As an aside, this sort of bounding can be easily done without any modification to the source code (i.e., using the code in Figure 7.5) with the help of the CLP metainterpreter defined in Chapter 9.

7.3.2 String and Integer Values are Not Bound

Another problem with the CLP typechecker code in Figure 7.5 is that emitted programs contain "holes", that is, uninstantiated portions. ASTs with holes cannot readily be used with testing, as holes represent portions of the AST which are incomplete. In this subsection, we deal specifically with two kinds of holes: those in `intValue`, and those in `strValue`.

Examples of holes in `intValue` and `strValue` can be seen from the output of the queries in Section 7.3.1, which contain underscores representing holes. Considering what the typechecker does and where these holes are, this behavior is relatively intuitive, if undesirable. While the typechecker reasons about string and integer types, it does not reason about particular string and integer values. As such, the moment it sees that it has reached a string or an integer, the typechecker does not need to proceed further; the typechecker already knows that it has an expression of type **string** or **integer**,

```
1  % decBound: Int, Int
2  decBound(In, Out) :-
3    In > 0,
4    Out is In - 1.
5
6  % TypeEnv: [pair(VariableName, Type)]
7  % typeof: Int, TypeEnv, Exp, Type
8  %
9  % VAR
10 typeof(_, Gamma, variable(X), Type) :-
11   member(pair(X, Type), Gamma).
12 % STRING
13 typeof(_, _, strValue(_), stringType).
14 % INTEGER
15 typeof(_, _, intValue(_), intType).
16 % + (int)
17 typeof(Bound1, Gamma, binop(E1, plus, E2), intType) :-
18   decBound(Bound1, Bound2),
19   typeof(Bound2, Gamma, E1, intType),
20   typeof(Bound2, Gamma, E2, intType).
21 % + (string)
22 typeof(Bound1, Gamma, binop(E1, plus, E2), stringType) :-
23   decBound(Bound1, Bound2),
24   typeof(Bound2, Gamma, E1, stringType),
25   typeof(Bound2, Gamma, E2, stringType).
26 % AFUN
27 typeof(Bound1, Gamma, afun(X, T1, E), funType(T1, T2)) :-
28   decBound(Bound1, Bound2),
29   typeOk(T1),
30   addMap(Gamma, X, T1, GammaPrime),
31   typeof(Bound2, GammaPrime, E, T2).
32 % ACALL
33 typeof(Bound1, Gamma, acall(E1, E2), T2) :-
34   decBound(Bound1, Bound2),
35   typeof(Bound2, Gamma, E1, funType(T1, T2)),
36   typeof(Bound2, Gamma, E2, T1).
```

Figure 7.6: Bounded version of the CLP code shown in Figure 7.5.

respectively. This follows directly from the typechecker's implementation in Figure 7.5, specifically in lines 8 and 10. Because underscore is used for the particular values in `strValue` and `intValue`, this effectively states that the value does not matter, and so holes end up being inserted into the resulting AST.

A quick and relatively simple approach to fixing these holes is to add additional constraints on the possible values held in `strValue` and `intValue`. This fix is shown in Figure 7.7. While this fix works, it suffers from two major problems. For one, this pollutes the typechecker code with information which is not concerned with typechecking; `validString` and `validInt` have nothing to do with typechecking, but they are nonetheless present in the typechecking code. A bigger, more subtle problem is that this can slow down generation, precisely because a non-typechecking concern was embedded into typechecking. To understand why, consider a context wherein a type **other** than **string** is required, but we do not know this ahead of time. (While we generally try to construct things to know this information ahead of time, this is sometimes unavoidable.) In such a situation, we may opt to generate a string constant (using the code starting at line 14 of Figure 7.7), only to later hit failure because we emitted the undesired **string** type. If a hole is used for the particular string constant, then backtracking will skip beyond the rule for string constants, because there are no choices to make for string constants when a hole is used. However, if a hole is **not** used, then we will end up backtracking right back to string constant, generating a **different** string constant in line 15 of Figure 7.7. While this clearly will still ultimately lead to failure (it still produces the unwanted **string**), there is no way for CLP to know this without proceeding forward with the new string constant. CLP will fruitlessly search in this manner until every string constant has been attempted, only at which point will backtracking try something other than a string constant. In short, by embedding particular constants into typechecking, this leads to a number of ultimately meaningless choices being considered, exploding the search space

in an uninteresting dimension and slowing down generation.

```
validString('foo').
validString('bar').

validInt(0).
validInt(1).

% TypeEnv: [pair(VariableName, Type)]
% typeof: TypeEnv, Exp, Type
%
% VAR
typeof(Gamma, variable(X), Type) :-
  member(pair(X, Type), Gamma).
% STRING
typeof(_, strValue(String), stringType) :-
  validString(String).
% INTEGER
typeof(_, intValue(Int), intType) :-
  validInt(Int).
% + (int)
typeof(Gamma, binop(E1, plus, E2), intType) :-
  typeof(Gamma, E1, intType),
  typeof(Gamma, E2, intType).
% + (string)
typeof(Gamma, binop(E1, plus, E2), stringType) :-
  typeof(Gamma, E1, stringType),
  typeof(Gamma, E2, stringType).
% AFUN
typeof(Gamma, afun(X, T1, E), funType(T1, T2)) :-
  typeOk(T1),
  addMap(Gamma, X, T1, GammaPrime),
  typeof(GammaPrime, E, T2).
% ACALL
typeof(Gamma, acall(E1, E2), T2) :-
  typeof(Gamma, E1, funType(T1, T2)),
  typeof(Gamma, E2, T1).
```

Figure 7.7: Version of the CLP code shown in Figure 7.6, with holes in `strValue` and `intValue` fixed.

For this reason, while the fix proposed in Figure 7.7 works, it is hardly ideal. A more efficient approach is to intentionally emit ASTs with holes in them, and then fill in the holes in a later pass. This keeps the search space for the typechecker as small as possible, and also segregates non-typechecker code elsewhere. An example of this sort of hole-filling pass is shown in Figure 7.8.

As an aside, the code in Figure 7.8 can be further refined with the help of the metainterpreter defined in Chaper 9. Specifically, the definitions of `validString` and `validInt` (in Figure 7.8) can be entirely elided from the source code, and treated as external parameters. Chaper 9 contains more relevant information.

```
 1  validString('foo').
 2  validString('bar').
 3
 4  validInt(0).
 5  validInt(1).
 6
 7  fillStringIntHoles(variable(_)).
 8  fillStringIntHoles(strValue(String)) :-
 9    validString(String).
10  fillStringIntHoles(intValue(Int)) :-
11    validInt(Int).
12  fillStringIntHoles(binop(E1, _, E2)) :-
13    fillStringIntHoles(E1),
14    fillStringIntHoles(E2).
15  fillStringIntHoles(fun(_, _, E)) :-
16    fillStringIntHoles(E).
17  fillStringIntHoles(anoncall(E1, E2)) :-
18    fillStringIntHoles(E1),
19    fillStringIntHoles(E2).
```

Figure 7.8: A major improvement over the hole-filling strategy presented in Figure 7.7, wherein holes are filled in a separate pass after typechecking occurs.

7.3.3 Variable Names are Not Bound

A third problem with the CLP typechecker code in Figure 7.5 which seems deceptively similar to the problem in Section 7.3.2 is that variable names are *mostly* unbound. The word "mostly" is used because while variables are never given particular names, they are nonetheless constrained, as in lines 5-6 and 20, 21 in Figure 7.5. Specifically, the intention is that two distinct variables may (though not necessarily) map to two distinct types in the type environment (i.e., `Gamma` in Figure 7.5). Conversely, the same variable is intended to always map to the same type in the type environment. Altogether, the intention is that the type environment is a mapping of variables to types, where the keys are all distinct.

It turns out that this intention of specifying a map is difficult to encode in CLP if the keys do not have determinate values, as with our variables and their unbound names. To illustrate the problem, assume that we have encoded the map as a listing of key/value pairs, as we have done in Figure 7.5. An example of such a map is shown below, where the variables `X` and `Y` are logic variables acting as placeholders for variable names:

```
[pair(X, intType), pair(Y, stringType)]
```

With the above map, we can trivially run into problems if we ever unify `X` and `Y`. Such a unification would merge these two variables into an equivalence class, and would effectively result in a multimap where the variable resulting from the unification of `X` and `Y` is nondeterministically of type `intType` *and* `stringType`. From a lexical scoping standpoint this is meaningless, as this ultimately means that the same variable is in scope *twice* with different associated types.

It is possible to avoid this problematic situation by carefully ensuring that we never perform such a unification of keys. Speaking from experience, however, this is much more difficult to ensure than it sounds, even with a carefully designed map interface. It also

tends to be painful to debug. As such, this is not considered a practical solution, which is consistent with advice given by O'Keefe [174].

Another solution is to add constraints which ensure that the keys must be distinct. To illustrate, consider the same map as before, duplicated below for convenience:

```
[pair(X, intType), pair(Y, stringType)]
```

Ultimately, if we could add an additional constraint that $X \neq Y$, that is, a disequality constraint between X and Y, then this problem would be solved. This can be done with the help of the arithmetic constraint solvers built into CLP, though it requires that variable names take on integral values. This would require a later conversion of integral variable names to non-integral variable names (e.g., `123`) to something more appropriate (e.g., `x123`). While this is all doable, speaking from experience, there tends to be a significant performance penality with this representation, and CLP libraries tend not to interact well with other solutions (particularly the metainterpreter, described in Chapter 9) thanks to upstream bugs in CLP engines themselves. As such, I have not seriously employed this solution.

The solution I tend to employ for this problem is the same sort of solution shown in Figure 7.7: define a procedure like `validVariable` which ensures that its argument is a valid variable name (whatever is appropriate for the situation), and be sure to call `validVariable` whenever a variable is encountered. This way, we ensure that keys will always be bound. The only area where care needs to be applied is when adding new key/value pairs to the map, which is a much easier and more localized concern which can be addressed entirely within the `addMap` procedure shown in Figure 7.5.

Unfortunately, this solution suffers from the same problems as those described in Section 7.3.3 for Figure 7.7: we add relatively useless choice points involving alternative variable names, bloating the search space in an uninteresting dimension. In practice,

this has not been a significant problem, only because the sets of variables considered has been small. If this were to become a problem, a worthwhile optimization is to have `validVariable` check to see if the variable is unnamed, and if not, then deterministically produce a random, possibly previously used variable name. Because of the deterministic nature of the optimized `validVariable`, this would not lead to fruitless choice points, though a variety of variable names are still possible. Such an optimization has not been necessary in my personal experience, though I have it in mind in case I ever observe this problem occurring.

7.3.4 Cyclic Terms are Possible

A fourth problem with the CLP typechecker code in Figure 7.5 is that it currently allows for the generation of *cyclic terms*. While this is a Prolog-centric problem, it can be explained purely in mathematical terms. For example, consider the following "type":

$$\tau = \tau \rightarrow \textsf{integer}$$

The word "type" is in quotes because this is clearly problematic: the type is self-referential, and is of infinite size. This self-referential behavior leads to some surprising results if we attempt to typecheck with it. For example, consider the following SimpleScala program, which uses τ:

$$(x : \tau) \Rightarrow x(x)$$

In the above program, an anonymous function's parameter was explicitly annotated with the faulty τ. The body of this function is suspect, calling its parameter x using x as a parameter. This mechanism can easily be exploited to perform general recursion using fixed-point combinators from the lambda calculus. However, this should not be possible: SimpleScala was intentially designed that recursion must occur through named functions,

but the above snippet uses no named functions. Even so, this code snippet typechecks according to the rules in Figure 7.3. A type derivation for this faulty program is provided in Figure 7.9.

$$\dfrac{\dfrac{}{\texttt{typeOk}(\tau)} \qquad \dfrac{\dfrac{}{[x \mapsto \tau] \vdash x : \tau \to \textbf{integer}}\,\textsc{var} \qquad \dfrac{}{[x \mapsto \tau] \vdash x : \tau}\,\textsc{var}}{[x \mapsto \tau] \vdash x(x) : \textbf{integer}}\,\textsc{ACall}}{[] \vdash (x : \tau) \Rightarrow x(x) : \tau \to \textbf{integer}}\,\textsc{AFun}$$

Figure 7.9: Type derivation for a simple program using the cyclic "type" τ, where $\tau = \tau \to \textbf{integer}$. Premises for the VAR rule show no surprising behavior, and so they have been omitted for space. At the point marked in **red**, the expected τ is exchanged for the equivalent $\tau \to \textbf{integer}$; this exchange is valid given the definition of τ.

When the CLP code is used as a typechecker, this situation is avoided by the fact that it is impossible for the user to ever annotate a type with our faulty τ; since τ has no finite representation, it simply cannot be written down as a normal type. However, this situation *can* occur when the CLP code is used as a generator, as CLP allows for the definition of cyclic terms. While cyclic terms have their uses in CLP, in a typechecking context this behavior is clearly undesirable; at the very least there is no way to even print out what a cyclic term is in a syntactically valid way (according to the syntax in Figure 7.1, and the call to `typeOk` would be non-terminating. Why CLP allows for cyclic terms is an artifact of the fact that unification in CLP, by default, does **not** perform the *occurs check* [175], which would otherwise prevent cyclic terms from being formed. The occurs check simply causes unification to fail at the moment we attempt to create a self-referential term.

To further illustrate the cyclic term problem, consider the following CLP snippet, representing the same construction for our aforementioned cyclic type τ:

```
?- Tau = funType(Tau, intType).
```

This snippet causes the logical variable `Tau` to be unified with the structure `funType(Tau, intType)`, which crucially contains `Tau`. This snippet succeeds, causing a cyclic term to be created. However, we can force the CLP engine to perform the *occurs check* during the unification with `unify_with_occurs_check`:

```
?- unify_with_occurs_check(Tau, funType(Tau, intType)).
```

The above snippet fails, because the engine would otherwise have to create a cyclic term.

With all this in mind, perhaps the most simplistic solution would be to force the engine to *always* perform the occurs check, even for a seemingly normal unification. Most engines, however, do not provide this capability. Moreover, there are legitimate uses of cyclic terms, and some CLP libraries depend on cyclic terms. As such, blanket enabling the occurs check would likely cause code to misbehave.

Another solution is to use `unify_with_occurs_check` (or its sibling `acyclic_term`, which fails if its parameter is a cyclic term) in any context wherein unifications on types are performed. While this solution is clearly safe, it has a major disadvantage that, in practice, this would mean hundreds of calls to `unify_with_occurs_check`, polluting code. Additionally, this would likely hurt performance significantly; engines intentionally do not perform the occurs check by default because it is expensive to perform, and the vast majority of unifications in practice cannot possibly form cyclic terms anyway. As such, we want to limit the number of calls performed to `unify_with_occurs_check` as much as possible, while still ensuring that cyclic types are never produced.

Based on my personal experiences, usually only one or two key calls to `unify_with_occurs_check` (or `acyclic_term`) are actually necessary to prevent cyclic terms from being formed. Historically, rules handling variable lookup along with rules handing the return types for potentially recursive functions tend to be the only points necessary where the occurs check is needed, though this depends somewhat on the par-

ticular type system in play. For example, with the SimpleScala typing rules in Figure 7.3, only the VAR and NCALL rules actually need the occurs check to be performed (specifically, VAR needs `unify_with_occurs_check` to unify the variable's type in the map with the expected return type, and NCALL needs `acyclic_term` to ensure that the return type for a named function is not cyclic). The fact that these two points (variable lookup and function return types) need the occurs check is not arbitrary: both allow code to refer to itself in some way (variables can be bound to something which is not yet fully known, which can later on be filled with something indirectly self-referential, and function return types can speak of recursive functions).

7.3.5 A Naive Generator is Born

Once the four fixes have been applied in Sections 7.3.1, 7.3.2, 7.3.3, and 7.3.4, the typechecker is able to function as a generator. However, the yielded generator is not particularly efficient, and it cannot realistically generate many interesting programs in a reasonable amount of time. Dramatic improvements are possible with some key optimizations, which can yield program generation rates of hundreds to thousands of programs per minute. These optimizations are discussed in Section 7.4.

7.4 Optimizing the Naive CLP-Based Generator for Well-Typed SimpleScala Programs

The generation rate of the generator yielded in Section 7.3 is poor, and is too slow to be a practical test case generator. The biggest reason why is that the generator will spend lots of time trying to search for nonexistent solutions. This is where CLP shines, because we can adjust exactly how the search is performed in order to avoid these nonexistent

solutions. Before we can adjust the search, however, we need to understand exactly *how* the search is misbehaving, and what needs to be done to correct it.

It turns out there are two distinct reasons in this case why we spend lots of time performing fruitless search. These are listed below, along with the subsections which are dedicated to explaining and fixing the problems:

1. Not all types are inhabited (i.e., some types have no corresponding programs of that type). Attempting to generate a program with an uninhabited type leads to fruitless search which will eventually fail, but only after exploring a state space of potentially exponential size. This issue is discussed further in Section 7.4.1.

2. The type we have been instructed to generate a program for may be too large to be generated with the current depth bound. For example, consider a type of depth d. In general, we would need a depth bound of at least d in order to generate such a type. If we instead have a bound d', where $d' < d$, then we cannot possibly satisfy the query and generate a program of such a type. This leads to fruitless search throughout the space defined by d', which is exponentially large. This issue is discussed further in Section 7.4.2.

7.4.1 Uninhabited Types

Not all types are inhabited in SimpleScala. As such, the typechecker may search a fruitless exponential space if it ever attempts to generate a program with an uninhabited type. For as, well, simple as SimpleScala is, there are surprisingly two different scenarios under which non-inhabited types can occur:

1. Bad (but legal) interaction with type variables

2. Meaninglessly cyclic **algebraic** definitions

Bad Type Variable Interactions

To see the problem with type variables, consider the following code snippet, where `???` is a stand-in for some expression:

```
def foo[A, B](a: A): B = ???
```

In the above snippet, it is not actually possible to instantiate `???` with any well-typed expression. Intuitively, the reason why is because there is nothing of type `B` in scope, and since `B` is a type variable, there is no possible way to derive something of type `B`. This follows directly from parametricity. [176]

In general, type variables cause major problems due to uninhabited types, and as far as I know there is not an easy solution to this problem. For example, it is neither sufficient nor necessary to ensure that all type variables used in the output of a function are introduced in the input. The following code snippet shows that this test is not sufficient:

```
def bar[A, B](f: A => B): B = ???
```

While the above snippet uses type variables `A` and `B`, it is not possible to derive something of type `B`, as this *also* requires derving something of type `A` in order to call the provied function from `A` to `B`. The following code snippet shows that such a type variable test is not necessary:

```
algebraic List[A] = Cons((A, List[A])) | Nil(Unit)
def baz[A, B](dummy: Int): List[(A, B)] = ???
```

In the above snippet, `???` can be instantiated to simply `Nil[(A, B)](unit)`, as we can always create an empty list parameterized by any type.

While I was able to devise a search-based solution to this problem, the search effectively was doing the exact same sort of work the typechecker was doing anyway, so it was

unlikely to see any actual benefit (i.e., it explored the same exponential space). As such, another solution was necessary to handle type variables in a better way. In my case, this was easy enough to restrict by forcing there to always be values in scope of all possible type variables in scope. This was handled by modifying exactly which expressions were valid within a **def**. For example, instead of generating the potentially problematic (where `???` is used to indicate a hole with arbitrary contents):

```
def foobar[A, B, C](x: ???): ??? = ???
```

...we instead generate the refinement:

```
def foobar[A, B, C](x: ???): A => B => C => ??? =
  (a: A) => (b: B) => (c: C) => ???
```

The variables `a`, `b`, and `c` do not overlap with variables which can be later introduced, and we do not allow nested **def**s in SimpleScala (unlike Scala). All these points collectively ensure that we will never hit a scenario where a type is uninhabited due to values of a given type variable not being present; we will always have `a`, `b`, and `c` in scope in the rest of the body of the **def**, and so we can always produce something of type `A`, `B`, or `C`. These values can be used to build up any other arbitrary type.

While this solution clearly limits the sort of programs we can generate (the previously-shown **def** `baz` snippet would never be emitted, for example), this was deemed an appropriate compromise. The generation rate is improved significantly with this change, and while it does change the kinds of programs generated (potentially leading to bugs missed which exist outside of this restricted subset of SimpleScala), the programs missed were considered unlikely to be of any more interest than those already being emitted. Usually an improvement in generation rate is worth this sort of compromise, but it depends on the particular problem.

Cyclic algebraic Definitions

Even without any type variables in play, SimpleScala also has uninhabited types, depending on the particular **algebraic** definitions in play. For example, consider the following definition (note that SimpleScala requires `[]` to be used to indicate that no type variables are used in a type or definition):

```
algebraic Uninhabited[] = Cycle(Uninhabited[])
```

It is not possible to derive any programs which are of the **algebraic** type `Uninhabited[]`, as the only constructor for `Uninhabited[]` itself requires a program of type `Uninhabited[]`, leading to an endless cycle. The depth bound will eventually kick in and stop this, but only after exhaustively searching fruitlessly through the whole space up to the depth bound.

Disallowing self-referential types is a viable solution to this problem, but it is severely limiting. For example, the usual linked list definition is self-referential (as previously shown), as well as any recursive data type. A better solution which is still relatively simple is to ensure that **algebraic** definitions contain a constructor with no user-defined types present. Because these sort of cycles can occur only with user-defined types, this enforces that there is always a cycle-free way to create an instance of a type. This is the solution employed in the actual generator. This is still limiting, though it allows for most common recursive data structures to be handled.

A more general solution is to explore all the **algebraic** definitions ahead of time and ensure that there are no cycles present, though this was deemed overly complex for the task at hand. In particular, because the simplistic solution can already generate the sort of programs of interest, it was not deemed necessary to put extra work in recovering the programs lost.

7.4.2 Types Which Exceed Bounds

In Section 7.3.1, a bound on program depth was added in order to make the search be over a finite space. This bound and the relatively simplistic way it is applied, however, can cause problems. To see why, consider the following type, which creates a triply-nested pair:

```
(Int, (Boolean, (String, Unit)))
```

A minimal program which satisfies this type follows:

```
(1, (true, ("foobar", unit)))
```

The problem is that the minimal program has depth 3, which means that the *only* way we can possibly generate a program of this type is if we have a depth bound of at least 3. If we have a smaller bound of, say, 2, then currently the generator in 7.3.1 would fruitlessly explore the *entire* exponential state space of programs of depth 2 or less before ultimately failing. This leads to severe hanging, even moreso than the problem of uninhabited programs (described in Section 7.4.1).

A key observation here is that this problem only arises when the type is known during generation but the program is not. This happens relatively frequently. For instance, if the generator chooses to call a function which has already been called at least once in the program, then the input type for the called function will have already been decided by the first call. In this case, subsequent calls must produce function inputs which match up with the expected (fixed) input type of the function. For subsequent calls, this results in a scenario where the type to generate is fixed (i.e., the input type of the function, needed to perform the function call), but the program (i.e., the actual expression evaluated when producing the input for the call) is not. If the type were not known (as with the first call to the function), then any arbitrary program can satisfy the generation task, including programs of depth 0 (e.g., **true**, **false**, 42, etc.).

Potential Solution: Fail as Soon as Depth is Untenable

The observation that this problem occurs only when the type is known hints at a solution: in contexts where the type is known but the expression is not, observe both the current depth bound and the actual type in play. If the type is deeper than the current depth bound, then immediately fail, which prevents searching through the fruitless state space. The idea here is to fail as soon as we determine that we cannot possibly satisfy the current type.

It turns out, however, that this solution does not work particularly well. For one, it is somewhat tricky to implement. At the very least, this solution requires us to ask if a logical variable holding a type is instantiated, meaning we must use built-in procedures like `var`, `nonvar`, and `ground`. These can quickly lead to code that is difficult to reason about and debug, which echoes warnings of O'Keefe [174]. We also need to potentially reason about *partially instantiated* types, not just fully known or fully unknown types. For example, we may be asked to generate the following type, where `???` is a hole:

```
(Int, ???)
```

In this case, we know that the minimal bound will be *at least* one, but it depends on exactly what `???`. We could fully instantiate the type at this point, though this is ill-advised; it is usually better to let the rest of the typechecker fill in types, as this ensures that generated programs will use the typing rules themselves for determining what to produce. Merely filling in a type here takes control away from the typing rules, and could potentially lead to a scewed program distribution.

In addition to being difficult to implement, this solution can occasionally cut off the search prematurely. As described, this modified search does not consider the contents of the type environment when choosing to cut off the search early. Depending on the variables in scope, it may be the case that there is, in fact, a smaller minimal program

possible. For example, consider the following code, where `???` is a hole which we want to fill in with a satisfying expression:

```
def small[](x: (Int, String)): (Int, String) = ???
```

The smallest possible program satisfying the expression in `small` (i.e., the `???` part) is simply `x`, the input to `small`. This program has depth 0. However, according to our search strategy, we need at least a program of depth one to satisfy this. As such, we might abort search early, as our strategy does not properly take variables into account.

Perhaps the biggest problem with this revised search is that fruitless exploration still occurs, though significantly less than before. The reason why is because this technique is based only on the *minimal* program size; larger programs are possible which satisfy the same type. For example, consider again the example type from the beginning of this section, along with the minimal satisfying program, which have been duplicated below:

```
1 (Int, (Boolean, (String, Unit)))
2 (1, (true, ("foobar", unit)))
```

While the above program on line 2 is minimal, it is hardly the only program that satisfies the type on line 1. For example, consider the following program:

```
(1 + 2 + 3, (true && false, ("ab" + "cd" + "ef", unit)))
```

This is a much larger program, but it still satisfies the type on line 1 of the previous listing. Indeed, for any inhabited type, the maximal program size satisfying that type is unbounded; trivially, we could always keep applying the polymorphic identity function, yielding a larger program without changing its type. Our revised search strategy would still cut generation off with such non-productive decisions, but it would cut it off only *after* the decision had been made. As such, we still end up exploring one level deep in the search space, which is still large (though not exponential). Moreover, we do this relatively frequently, so significant amounts of fruitless search still occur overall.

Solution: Ensure Progress Towards Target Type when Bound is Reached

Overall, trying to full-on fail early when the bound is untenable is not a very good solution in practice. The solution actually employed takes advantage of a key property afforded to us thanks to the work in Section 7.4.1: with those modifications, all types are inhabited in SimpleScala. As such, we know that *some* program is always possible for any possible type. Moreover, looking closely at the typing rules, we can see that some rules will always build bigger types from smaller types (e.g., AFun), and that those rules collectively handle all types in play. (From a types-as-logic perspective [46], the introduction rules in the SimpleScala type system are of particular interest to us.) With all this in mind, a much more sophisticated solution is possible: when the bound is reached, enforce that we only use those rules which make progress towards the particular type it is we want to generate. Because types are finite (thanks to ensuring we will not have any cyclic terms in Section 7.3.4), and all types are inhabited, this will guarantee that we will eventually generate a program of the expected type, even if the bound is completely ignored. While this turns the bound into more of a suggestion than a hard limit on program size, in practice this is irrelevant because the bound is just needed to keep things finite (see Section 7.3.1).

Implementing this change requires a fairly dramatic restructure of the typechecker code. While there are surely different ways to implement this, speaking from personal experience, the simplest way was to separate rules that build up types (i.e., introduction rules) from those that break down types (i.e., elimination rules), and then call the appropriate set of rules when necessary. Doing this easily required breaking the inherent recursion in the rules, so the code was restructured to produce *delayed* constraints. That is, instead of performing a recursive typechecking call, the generator would instead produce a data structure representing the result of the recursive call, without actually

performing the call. This data structure acted as a delayed constraint. These delayed constraints could then be executed later.

To see this in practice, we will start from the bounded typechecker shown in Figure 7.6. We first split the original `typeof` into `typeofIntroduction` and `typeofElimination`, shown in Figure 7.10. These two rules are no longer recursive, and instead return delayed constraints. We then update the original `typeof` accordingly, which now calls either `typeofIntroduction` or `typeofElimination` and processes any returned delayed constraints. This updated `typeof` definition is shown in Figure 7.11.

As shown in Figures 7.10 and 7.11, the logic of the typechecking rules is no longer contained in `typeof` (as it is in Figure 7.6), but is instead spread across the `typeofIntroduction` and `typeofElimination` rules. The `typeofIntroduction` and `typeofElimination` rules now return `delayed` structures, which contain everything necessary to perform a recursive call. It is now the responsibility of `typeof` to perform the actual recursive calls. In order to perform these recursive calls, the `typeof` procedure will first gather some delays using some combination of `typeofIntroduction` and `typeofElimination`. From here, `typeof` will indirectly recursively call itself through the `processDelays` helper, which iterates through the given list of delays and calls `typeof` on each.

The most important change to bring out in Figure 7.11 is that if the bound hits `0` (on line 3), then **only** introduction rules are considered (in the body of the first rule of the `typeof` procedure, starting at line 4). If both the expression and type is unconstrained (on line 4), then `typeof` will select a base case for the expression and type and stop recursing (on line 7). Otherwise, `typeof` will recurse, but only with introduction rules (on lines 8-9), which guarantee progress towards whatever type is in play. (As written, this deviates slightly from the previous discussion, as it could also be the case that the expression is known but the type is not, in which case we may spuriously mark a known

```
1  % typeofIntroduction: Gamma, Exp, Type, Delays
2  %
3  % VAR
4  typeofIntroduction(Gamma, variable(X), Type, []) :-
5          member(pair(X, Type), Gamma).
6  % STRING
7  typeofIntroduction(_, strValue(_), stringType, []).
8  % INTEGER
9  typeofIntroduction(_, intValue(_), intType, []).
10 % AFUN
11 typeofIntroduction(Gamma, afun(X, T1, E), funType(T1, T2),
12                      [delayed(GammaPrime, E, T2)]) :-
13   typeOk(T1),
14   addMap(Gamma, X, T1, GammaPrime).
15
16 % typeofElimination: Gamma, Exp, Type, Delays
17 %
18 % + (int)
19 typeofElimination(Gamma, binop(E1, plus, E2), intType,
20                    [delayed(Gamma, E1, intType),
21                     delayed(Gamma, E2, intType)]).
22 % + (string)
23 typeofElimination(Gamma, binop(E1, plus, E2), stringType,
24                    [delayed(Gamma, E1, stringType),
25                     delayed(Gamma, E2, stringType)]).
26 % ACALL
27 typeofElimination(Gamma, acall(E1, E2), T2,
28                    [delayed(Gamma, E1, funType(T1, T2)),
29                     delayed(Gamma, E2, T1)]).
```

Figure 7.10: The core logic of the `typeof` rule from Figure 7.6, refactored into two procedures: `typeofIntroduction` and `typeofElimination`. Instead of making recursive calls, `delayed` data structures are returned which encode typechecking constraints which are delayed until later.

expression as ill-typed if the expression is not in the introduction rules. However, in practice this situation is not observed.) Conversely, if the bound has **not** been reached in the `typeof` procedure in Figure 7.11 (starting on line 10), then we can select from either elimination rules (line 13) or introduction rules (line 14), and then recursively move on

```
1  % typeof: Int, TypeEnv, Exp, Type
2  % The Int is the bound
3  typeof(0, Gamma, Exp, Type) :-
4    ((var(Exp), var(Type)) ->
5      % If we are completely unconstrained, then pick
6      % some base case and terminate
7      (typeofIntroduction(Gamma, Exp, Type, []));
8      (typeofIntroduction(Gamma, Exp, Type, Delays),
9       processDelays(Delays, 0))).
10 typeof(N, Gamma, Exp, Type) :-
11   N > 0,
12   % select elimination OR introduction
13   (typeofElimination(Gamma, Exp, Type, Delays);
14    typeofIntroduction(Gamma, Exp, Type, Delays)),
15   NewN is N - 1,
16   processDelays(Delays, NewN).
17
18 % processDelays: [Delay], Bound
19 processDelays([], _).
20 processDelays([delayed(Gamma, Exp, Type)|Delays], Bound) :-
21   typeof(Bound, Gamma, Exp, Type),
22   processDelays(Delays, Bound).
```

Figure 7.11: The updated `typeof` rule from Figure 7.6, which now uses the `typeofIntroduction` and `typeofElimination` procedures defined in Figure 7.10.

(line 16).

A side advantage of this strategy of breaking the recursion in the typechecking rules themselves is that the typechecking rules no longer contain explicit bounds. That is, `typeofIntroduction` and `typeofElimination` in Figure 7.11 are not recursive, so there is no need to place bounds on them. In contrast, `typeof` is recursive, so a bound is needed. However, `typeof` is much smaller than it was in Figure 7.6, so there is overall much less modification needed to deal with bounding. Moreover, the bounding is now intrinsically tied to the generation itself. When the bound is reached, we no longer simply fail, but instead fundamentally change the behavior of the generator and proceed

forward.

There is one last point of interest with Figures 7.10 and 7.11. In the actual source code, `typeofIntroduction` takes an auxilliary bounding-related parameter, which merely records if the bound has been reached. An edge case in assuming that all types are inhabited is that if we are asked to produce a program of a user-defined type (i.e., an **algebraic**), we can *still* end up in infinite recursion. To understand why, consider the standard definition for lists, which are recursive:

```
algebraic List[A] = Cons((A, List[A])) | Nil(Unit)
```

This definition does not contain cycles, and is compatible with the cycle-breaking behavior described in Section 7.4.1; there is a single constructor defined which does not utilize user-defined types (**algebraic**s), namely `Nil(Unit)`. Consider a context under which we must produce a program of type `List[Int]`, and the bound has been reached. This will enforce that we choose the rule which will introduce a user-defined type, namely CONS (short for "constructor") in Figure 7.3. This rule would have been placed in `typeofIntroduction` in Figure 7.10, in order to ensure the rule could be used when the bound was reached. However, from looking at the list definition, there is clearly a problem. Without some sort of extra guidance, we can always select `Cons` instead of `Nil`, which would put generation in an infinite loop. As such, an extra bounding-related parameter is passed along to `typeofIntroduction`, just for this specific purpose. If the bound is reached, the parameter forces the code implementing CONS to choose constructors which do not use user-defined types, forcing `Nil` to be used instead of `Cons` in the above example. This modification guarantees we will not loop on generating user-defined types, again ensuring progress towards whatever the current type is.

7.5 Results

A full generator for SimpleScala was produced, ultimately yielding something that looked much like the generation code presented in Section 7.4. This generator was able to produce about 200 well-typed programs per minute underneath the metainterpreter (described in Chapter 9), which was utilized primarily to add random search capabilities. While this is still relatively slow compared to a number of other generators I have developed, this is still orders of magnitude faster than the relatively advanced work of Fetscher et al. [42], which is the closest work we can compare to. Additionally, the metainterpreter alone comes with severe performance costs (described in Section 9.6).

Major opportunities for optimization are still present in this generator, revealed by the fact that the generation rate is not very consistent; it will produce programs in spurts as opposed to a steady stream of programs, indicating that there are still code paths which fruitlessly search the state space. However, at 200 programs per minute, this was fast enough to be able to generate a sizable test suite for the corresponding CS162 assignment within a reasonable amount of time, so no further optimizations were performed.

Overall, this generator was used to produce over 100K programs within several hours, which served as the main internal test bank for the assignment. I did not pursue IRB approval to study how effective this generator was for testing student code, so I cannot comment on the bug-finding effectiveness of this test suite.

7.6 Conclusion

This chapter has explored a very complex generator in-depth, and it has similarly shown how such a complex generator can incrementally arise from a more simplistic

generator. While this process is certainly involved and difficult at times, the end result is a generator which is orders of magnitude faster than competitors, with remaining room for optimization. Despite the fact that no industrial-grade software was tested with this generator, this was the most sophisticated generator I have ever worked on by far, both from a theoretical level (generating parametric polymorphic, generic programs is challenging even in theory) and a pragmatic level (there are a multitude of coding challenges, and the code transformation described in the latter portion of Section 7.4.2 is arguably a novel Prolog design pattern). Thanks to this generator, a test suite consisting of over 100K programs was generated for a complex CS162 assignment within a matter of hours. The programs in the test suite are sizable and perform a large variety of complex operations, and so these should give student solutions a good stress test.

Overall, this serves as an excellent case study to bolster my thesis. It would not have been possible to encode these test cases at all if not for the fact that CLP was so expressive. Additionally, with significant optimization, CLP was able to generate programs fast enough to form an effective test suite within a reasonable amount of time. Such optimizations were only possible because CLP gives the programmer fine-grained control over how constraint search is performed, allowing me to take advantage of domain-specific properties like whether or not a type is inhabited. Among the constraint solvers discussed in Chapter 1, Section 1.6, this capability to control search is unique to CLP; other constraint solvers would simply be too slow for the problem, if the problem could even be encoded at all. This all serves as evidence that CLP is an excellent solution to the generalized structured black-box test input generation problem.

Chapter 8

Improving CLP for Testing: Typed-Prolog

8.1 Introduction and Motivation

In this chapter, a type system and related language for CLP is discussed. While this chapter does not contribute directly to my thesis' argument (that is, this chapter does not speak to the generality, expressibility, or performance of CLP when applied to structured black-box test case generation), I have used it with great success to implement a wide variety test case generators. With this in mind, this chapter is not about strengthening the thesis, but rather about discussing closely related technologies and insights for practical usage of the insights and contributions of this thesis.

While CLP has proven itself to be immensely useful for automated test case generation, it is still imperfect for this task. In this chapter, there are three particular problems of concern I discuss and address:

1. CLP largely lacks a notion of types

2. CLP does not have proper capabilities for handling something like a functional higher-order function, and overall abstracts over computation poorly

3. CLP lacks a portable, sane module system

To elaborate further on the first problem, CLP is not a statically-typed language, nor can it be properly even qualified as a dynamically-typed language. For this reason, bugs can very easily slip into CLP-based test case generators. These bugs are particularlly challenging to debug in the CLP setting. Moreover, given the many tests generated, it may take a significant amount of time and resources before it becomes apparent that there even *is* a bug present. This is because bugs often manifest as failing to produce a particular program of interest, which is difficult to check for, especially given the often vague definition of "interest". This problem of bugs is elaborated on in Section 8.3.1.

As for the second aforementioned problem, it is a hindrance that CLP lacks proper support for abstracting over computation, as with higher-order functions. This leads to repetitive code, and it makes it difficult to port ideas from functional languages into CLP. Because both functional languages and CLP share the same idiom stressing pure core (i.e., code without side-effects like mutable state), this can be frustrating. Moreover, the idiomatic CLP "solution" to abstracting over computation entails the dynamic construction of programs, and fundamentally is no different from performing frequent calls to an `eval`-like routine (as seen in Lisp, JavaScript and Python). This entails all the usual problems of `eval`, including debugging difficulties and severe performance penalties. This problem is further discussed in Section 8.3.2.

With the third problem, it is far more difficult than it should be to split up CLP code into multiple files with distinct namespaces. There is no standardized CLP module system [177], so different implementations solve this problem in different, sometimes incompatible ways. Indeed, even relatively popular implementations like GNU Prolog [66]

completely lack any sort of module system [177], making it difficult to port the same CLP code to different engines. More discussion of this problem follows in Section 8.3.3.

In order to solve these problems, I propose a new language which exists as a thin wrapper on CLP: Typed-Prolog. Typed-Prolog features a type system based on Hindley-Milner [178, 179], solving the first problem via static types and static typechecking. A key type in Typed-Prolog is that of higher-order *relations*, which occupy a similar place as higher-order functions and have a similar look and feel. Typed-Prolog also features a simple, but effective, module system which can handle the import and export of code and data between files. Arguably the best part about Typed-Prolog is that it compiles down to standard CLP *without* the use of any `eval`-like built-in procedures, even if higher-order relations are used. This ensures maximum portability across different engines, without any significant performance costs.

Overall, this chapter makes the following major contributions:

- A thorough discussion of the three aforementioned problems, in the context of automated test case generation (in Section 8.3.3)
- A formalization of the type system employed to solve the typing problem (in Section 8.4)
- A discussion of how higher-order relations are handled in Typed-Prolog (in Section 8.5)
- A discussion of how the module system works in Typed-Prolog (in Section 8.6)
- A discussion of how Typed-Prolog has been applied to test case generation, and the sort of advantages it has provided over plain CLP (in Section 8.7)

8.2 Related Work

Previous chapters lacked their own related work chapters, reflecting the fact that the related work for the overall thesis (Chapter 1, Section 1.3) was redundant with any chapter-specific related work. However, given that this chapter is more about a better CLP than using CLP for testing, the related work for this chapter is radically different than for prior chapters. As such, this chapter features its own separate related work section.

Mycroft et al. [180] first studied the application of the polymorphic Hindley-Milner type system [179, 178] to Prolog. This was powerful enough to encode polymorphic lists and operate over them in a similar way to that of Typed-Prolog. While Mycroft et al. discusses "higher-order objects" which bear close resemblance to Typed-Prolog's higher-order relations, these are only discussed from a high level, and are not included with the main type system or its implementation.

O'Keefe mentions that a variety of type systems are available for CLP [174], though none of these are available in any modern engine I am aware of (the source is from 1990). A search of type systems for Prolog from this time revealed a number of unsound type systems. For example, Lakshman et al. [181] discuss a type system which is supposedly based on Mycroft et al. [180], though upon closer examination the type system is unsound; it supposedly handles full metaprogramming features including `assert` and `retract` in a static manner, which is fundamentally impossible without additional restrictions. It is questionable what utility fundamentally unsound type systems have, given that these cannot reliably used to catch bugs in CLP code ahead of time.

While work on adding type safety to CLP itself seems to have waned, there are other logical languages in existence with sound type systems. Both Mercury [182] and Curry [183] are based on the Mycroft-O'Keefe [180] system, and they augment it with

typeclasses [184, 107] and capabilities equivalent to the higher-order relations in Typed-Prolog. Mercury additionally features advanced mode analysis which can be used to automatically avoid the inefficient "generate-and-filter" style [185]. While these features make Mercury and Curry look appealing for test case generation, they are not well-suited to this task. Mercury is prohibitely restrictive in what it allows you to write, and forces the programmer to bare a significant type annotation burden. I briefly used Mercury for test case generation purposes, until it became apparent that a relatively simple generation task was not possible to encode in Mercury without repeatedly escaping from Mercury to another language. Curry has no standard implementation, and none of the existing implementations are built with performance in mind; this is in stark contrast to modern CLP engines. Additionally, the semantics behind Curry are relatively complex compared to other logical languages [186], further hindering adaptability.

8.3 Problems with CLP for Test Case Generation

This section describes in detail three key problems in using CLP for test case generation. The first of these problems is that CLP lacks a strong notion of types, which is subsequently discussed.

8.3.1 CLP Largely Lacks Types

While CLP is certainly not a statically-typed language, it is arguably not a dynamically-typed language, either. For example, consider the CLP code snippet in Figure 8.1. This snippet contains a simple but high-impact bug: in line 1, it is expected that `int` is of `intType`, **not** the typo `intTyype`.

In a statically-typed language, one would expect the typechecker to reject the buggy program in Figure 8.1 ahead of time. Because no such rejection occurs (i.e., CLP has

```
1 typeof(int(_), intTyype).
2 typeof(bool(_), boolType).
3 typeof(plus(E1, E2), intType) :-
4   typeof(E1, intType),
5   typeof(E2, intType).
6 typeof(and(E1, E2), boolType) :-
7   typeof(E1, boolType),
8   typeof(E2, boolType).
```

Figure 8.1: CLP snippet which is intended to handle the typing rules of a simple arithmetic language. The language handled features integers (`int`), booleans (`bool`), arithmetic addition (`plus`), and boolean conjunction (`and`). A typo is present in line 1, where `intTyype` is used instead of `intType`.

no built-in typechecker), CLP is clearly not statically-typed. In a dynamically-typed language, one would expect some sort of runtime type error once line 1 was encountered, but similarly CLP produces no such runtime error. Instead of producing a runtime error, the CLP code ends up behaving in a completely unexpected way: `plus` is always flagged as ill-typed if used as a typechecker, and `plus` is never emitted if used as a generator. The root cause for this behavior is that `int` effectively has type `intTyype`, *not* the expected `intType`. Since `plus` needs its operands to be of `intType` and currently no base case in the rules exists to produce something of `intType` (i.e., the only rule that produces `intType` itself is the recursive rule handling `plus`), `plus` can never possibly be emitted.

Generalizing from the example in Figure 8.1, arguably type errors usually manifest with unexpected behavior in CLP, as opposed to outright program rejection (in a statically-typed setting) or a runtime error (in a dynamically-typed setting). Specifically, the behavior manifests as *unexpected unification failure*. To better understand this, line 4 of the example recursively calls `typeof` with `E1`, and forces the return type of `typeof` to unify with `intType` (i.e., `E1` must have type `intType`). For the recursive call, while rule 1 may be considered, it will not execute, because `intType` (passed in the recursive call at line 4) does not unify with `intTyype` (present in line 1). In terms of expected

code behavior, this unification was expected to succeed, hence I refer to the unification failing as "unexpected unification failure".

The problem of unexpected unification failure is, in practice, very difficult to debug. Even in the small example in Figure 8.1, we can see non-local behavior triggered by this bug: while the actual bug is present in line 1, we instead see that the code in line 3 never gets executed, despite the fact that there are no problems with the rule in lines 3-5. This sort of non-local behavior means that lots of code can become suspect in the presence of unexpected unification failures. Additionally, since unification failure and failure overall are normal parts of CLP execution, isolating the unexpected failures from the expected failures amounts to finding the proverbial needle in the haystack.

Perhaps worst of all is that merely identifying unexpected unification failure can be difficult. The only reason this behavior was readily apparent in Figure 8.1 is because the code is of low complexity, making it easy to identify code misbehavior. In practice, trying to manually spot such misbehavior is infeasible with complex generators, and even with automation this is a challenge. Speaking from experience, I have seen a generator run for two entire weeks before it was noticed that it never produced a particular kind of test, where the test of interest exhibited a conjunction of properties which all worked in isolation. Sure enough, there was a typo much like the one present on line 1 of Figure 8.1, which ultimately prevented tests from being generated with the conjunction of properties.

8.3.2 CLP Lacks Proper Support for Higher-Order Functions

As with functional programming, CLP puts a strong emphasis on writing *pure* code; that is, code that lacks mutable state and other side-effects. While this can be restrictive, in a functional programming setting this is somewhat alleviated by the presence of higher-order functions, which serve as a fundamental abstraction. However, CLP does not

contain any features which behave like true higher-order functions. This can lead to highly repetitive code.

The idiomatic CLP solution to this problem is to use *metaprogramming* in order to dynamically construct a procedure call. These calls can then be called with the `call` built-in, which serves the same role as the `eval` operation from dynamic scripting languages (e.g., JavaScript, Ruby, and Python). These dynamically constructed calls can be used to abstract over computation, which is fundamentally what a higher-order function allows. To illustrate how this works in CLP, consider the code snippet in Figure 8.2. As shown, both `multEachByFive` and `addToEach` have the same basic structure of performing a predefined operation on each element of a input list, yielding an output list of the same length which holds the result.

```
1  multEachByFive([], []).
2  multEachByFive([N|Ns], [M|Ms]) :-
3    M is N * 5,
4    multEachByFive(Ns, Ms).
5
6  addToEach(_, [], []).
7  addToEach(Amount, [N|Ns], [M|Ms]) :-
8    M is N + Amount,
9    addToEach(Amount, Ns, Ms).
```

Figure 8.2: Two procedures which perform an operation on an input list, yielding a new list. `multEachByFive` multiplies each element of a input list by five, yielding an output list holding the result. `addToEach` adds a given amount (the first parameter) to each element of an input list, yielding an output list holding the result.

This sort of pattern of needing to apply an operation to each element of some list arises quite frequently both in CLP and in functional programming. From the functional programming setting, this pattern is commonly abstracted over by the `map` function. The `map` function takes an input list along with the actual computation to perform, represented with a higher-order function. With these inputs, `map` will apply the computation to each

element of the input list, ultimately returning an output list. To see how this works in a functional setting, the code above has been reimplemented to use `map` in Scala in Figure 8.3, and a custom `map` definition has been similarly provided (`map` is predefined in Scala, but part of the intention of this example is to show how `map` can be defined).

```
1  def map[A, B](list: List[A], f: A => B): List[B] = {
2    list match {
3      case head :: tail => {
4        f(head) :: map(tail, f)
5      }
6      case Nil => Nil
7    }
8  }
9
10 def multEachByFive(list: List[Int]): List[Int] = {
11   map(list, (x: Int) => x * 5)
12 }
13
14 def addToEach(amount: Int, list: List[Int]): List[Int] = {
15   map(list, (x: Int) => x + amount)
16 }
```

Figure 8.3: How the `map` operation can be implemented in Scala, along with applications of `map` to implementing the `multEachByFive` and `addToEach` procedures originally defined in Figure 8.2. These procedures have been ported to Scala.

As shown in Figure 8.3, `map`'s implementation in Scala is heavily dependent on the presence of higher-order functions; `map` takes a higher-order function (the second parameter in line 1), and calls to `map` need to pass higher-order functions on lines 11 and 15. While CLP lacks native higher-order functions, we can still make do with the help of `call`. To illustrate how this works, the Scala code in Figure 8.3 has been ported to the closest CLP equivalent in Figure 8.4, which takes advantage of `call` and the dynamic construction of procedure calls.

As shown in Figure 8.4, the actual implementation of `map` arguably does not differ

```
 1 map([], _, []).
 2 map([A|As], AToB, [B|Bs]) :-
 3     call(AToB, A, B),
 4     map(As, AToB, Bs).
 5
 6 doMultByFive(N, M) :-
 7   M is N * 5.
 8
 9 multEachByFive(Input, Output) :-
10     map(Input, doMultByFive, Output).
11
12 doAddAmount(Amount, N, M) :-
13   M is N + Amount.
14
15 addToEach(Amount, Input, Output) :-
16     map(Input, doAddAmount(Amount), Output).
```

Figure 8.4: Closest direct port of the Scala code in Figure 8.3 to CLP, using the built-in `call` procedure of CLP.

significantly from the implementation of `map` in Figure 8.3. Both variants return the empty list if the input is the empty list, and this serves as the base case for the recursion. For the recursive case, if the list is non-empty, then the computation is applied to the first element of the list, yielding the first element of the result list. A recursive call to `map` is then performed in both the CLP and Scala code in order to process the rest of the elements in the input list, yielding the rest of the output list.

However, while the definition of `map` is quite similar in both Figure 8.3 and Figure 8.4, the actual *usage* of `map` is very different. Because CLP lacks higher-order functions, we cannot easily pass in a whole computation, as we can in Figure 8.3 in lines 11 and 15. Instead, we must define a separate procedure for each computation we intend to pass in this style, yielding `doMultByFive` and `doAddAmount` in Figure 8.4. We then pass this computation by either passing the name of the procedure to call (as is done in line 10), or passing a structure with the name of the procedure to call along with the first arguments

needed (as is done in line 16). When `call` is finally reached, if it is given the name of a computation (e.g., `doMultByFive`), it will first construct a structure with the given name, along with whatever arguments it was passed. For example, when `doMultByFive` was ultimately passed to `call` in line 3 (originally passed to `map` in line 10), `call` ends up assembling a structure that looks like the following:

```
doMultByFive(A, B)
```

The `call` built-in will then take this assembled structure and call it as if it were a normal procedure call. A similar process is followed if `call` is passed a structure, except in the structure case the arguments to the structure are treated as the first parameters to the procedure to call. For example, when `doAddAmount(Amount)` was ultimately passed to `call` in line 3 (originally passed to `map` in line 16), `call` ends up assembling a structure that looks like the following:

```
doAddAmount(Amount, A, B)
```

This assembled structure is similarly called as if it were a normal procedure call.

While this `call`-based mechanism of abstracting over computations works, it suffers from a number of problems. For one, as shown in Figure 8.4, we had to add two extra procedures which were not present in the Scala equivalent in Figure 8.3, namely `doMultByFive` and `doAddAmount`. These are pure boilerplate, and are subject to careless errors as a result. A second problem, illustrated by `doAddAmount` in line 12, is that the ordering of parameters in clauses becomes an important issue. Specifically with `doAddAmount`, the `Amount` parameter on line 12 *must* be the first parameter. Otherwise, the `Amount` parameter passed in line 16 would have completely different meaning, specifically whatever the first parameter of `doAddAmount` was. (In this case, the point is relatively moot considering what `doAddAmount` does, but this cannot be expected to hold in general.) A third problem is that it is easy to forget to pass a parameter or pass too

many parameters, in which case a runtime error will occur when `call` is used. Effectively, uses of `call` are dynamically typed, leading to errors which are non-local in nature (i.e., usually the `call` usage is correct, but something passed into `call` was malformed, which is a problem elsewhere in the code). Lastly, the use of `call` leads to steep performance penalties, ultimately because it is so difficult to optimize. These penalties can range anywhere from around 3× slower (based on prior benchmarks done in SWI-PL [67]), to 14× slower (based on prior benchmarks done in GNU Prolog [66]).

8.3.3 CLP Lacks a Sane Module System

The capability to split code into multiple files with coherent rules about namespacing is a basic capability any language needs in order to develop large codebases. This capability, however, does not exist in CLP in any standardized form [177]. Different engines take different approaches to implementing module systems, and some make no attempt whatsoever (e.g., GNU Prolog [66, 177]). This lack of standardization discourages CLP programmers from using any sort of module system, lest they have to spend extra time porting their code from one engine to another.

A root cause for this lack of a standardized module system is precisely because of the `eval`-based approach used for parameterizing computation, discussed in Section 8.3.2. The discussion in that section was dependent on all callable procedures existing in the same namespace, which is precisely the sort of situation we wish to avoid with the proper use of a module system. With modules, it becomes possible to pass computation names and structures between modules. At the point where a call is made, we must know exactly which module is being called into, in addition to the actual procedure to call. This alone is not a trivial matter to resolve, and it is made more complex by the fact that a parameterized computation may itself contain a parameterized computation. While this

may seem strange, this sort of phenomenon can easily come about with code written in a functional style.

8.4 Type System

In this section, I formally define the type system used by Typed-Prolog.

8.4.1 General Design and Syntax

While Typed-Prolog overall contains a number of components, only a subset of these interact with the typing rules. From a high-level, these components are as follows:

- Built-in support for atoms, integers, and generic lists.
- Handling for making and calling higher-order relations. Syntactic higher-order relations are considered a kind of term, and behave like data in this way.
- Handling for calling named procedures, which have been annotated with types by the user.

The abstract syntax for these various components is shown in Figure 8.5.

8.4.2 Type Domains

Before discussing the actual domains and formal definitions used throughout type-checking, it is necessary to understand a basic capability of Typed-Prolog which influences the design of these domains.

$$x \in \mathit{Variable} \qquad a \in \mathit{Atom} \qquad i \in \mathbb{Z}$$
$$\mathit{cln} \in \mathit{ClauseName} \qquad \mathit{cn} \in \mathit{ConstructorName} \qquad \mathit{un} \in \mathit{UserDefinedTypeName}$$
$$T \in \mathit{TypeVariable}$$

$$\begin{aligned}
\tau \in \mathit{Type} &::= \textsf{int} \mid \textsf{atom} \mid \textsf{bool} \mid \textsf{relation}(\vec{\tau}) \mid \mathit{un}(\vec{\tau}) \mid T \\
\oplus \in \mathit{Binop} &::= + \mid - \mid \times \mid \div \\
\otimes \in \mathit{Unop} &::= - \mid \textsf{abs} \\
\mathit{lhs} \in \mathit{LeftHandSide} &::= x \mid i \\
e \in \mathit{Exp} &::= x \mid i \mid e_1 \oplus e_2 \mid \otimes e \\
\mathit{term} \in \mathit{Term} &::= x \mid i \mid a \mid \textsf{lambda}(\overrightarrow{\mathit{term}}, \mathit{body}) \mid \mathit{cn}(\overrightarrow{\mathit{term}}) \\
\mathit{bbinary} \in \mathit{BodyBinaryOp} &::= \wedge \mid \vee \mid \implies \\
\ominus \in \mathit{CompareOp} &::= < \mid \leq \mid > \mid \geq \\
\mathit{body} \in \mathit{Body} &::= \mathit{lhs}\ \textsf{is}\ e \mid \neg \mathit{body} \mid \mathit{body}_1\ \mathit{bbinary}\ \mathit{body}_2 \mid e_1 \ominus e_2 \\
&\quad \mid \textsf{call}(\mathit{term}_1, \overrightarrow{\mathit{term}_2}) \mid \mathit{cln}(\overrightarrow{\mathit{term}})
\end{aligned}$$

Figure 8.5: Abstract syntax of the portion of Typed-Prolog that interacts with Typed-Prolog's type system. All sequences of elements are permitted to be empty (i.e., we may perform a call without any parameters). Certain liberties have been taken with this syntax that differ from the concrete syntax. Most arithmetic operators have been omitted as they are uninteresting from a type standpoint. $\wedge$, $\vee$, $\implies$, and $\neg$ are used to represent "`,`", "`;`", "`->`", and "`\+`", respectively.

Variable Scoping in Typed-Prolog

Like CLP, Typed-Prolog allows for variables to be introduced in a number of contexts. For example, consider the following unification:

```
X = foo(bar(Y), Y).
```

The variable `X` is introduced first, which is immediately unified with the `foo` structure. From here, the variable `Y` is introduced in `bar`, which is later re-used in the second parameter to `foo`. While this example is small, it demonstrates that a basic lexical scoping approach does not really work for Prolog; variables can be introduced in a deeply

nested term (e.g., `bar` in the example above), which can escape to an outer term (e.g., the second value for `foo` in the example above).

The fact that lexical scoping does not quite work is significant to our typing rules, as the typical approach of only threading the type environment down (as was done in the rules for SimpleScala in Section 7.2) will not be sufficient; lower scopes can change the type environment, and these changes need to be propagated back up. We *could* fix this by translating the original code into a lexically-scoped form. However, based on personal experiences, the resulting translation is itself significantly complex, and this sees diminishing returns if the overall goal is to reduce complexity. As such, we instead modify the typing rules themselves to pass a type environment up, in addition to taking an input type environment. The notation we use for this is as follows:

$$\Gamma_{in} \vdash e : \tau \cdot \Gamma_{out}$$

The above notation states that the type of e should be τ underneath the type environment Γ_{in}. In addition, the typechecking of e yields the output type environment Γ_{out}.

Type Domains and Definitions

The typing rules for Typed-Prolog utilize a number of formal definitions which are provided in Figure 8.6. A brief description of each one of these definitions follows:

- *clauses*: Records named procedures which can be called (as name/arity pairs), along with the type variables each introduces and the expected types of the clause.
- *tdefs*: Records user-defined type definitions. This is maintained as a mapping of constructor name/arity to a 3-tuple of the following:
  - The name of the user-defined type the constructor belongs to
  - The type variables introduced by the user-defined type

– The expected types for the constructor parameters

- Γ: The type environment, which maps variables to their expected types. In the implementation, each mapped type is actually a logical variable representing a type; that is, types may be known, unknown, or partially known at any time thanks to type inference and unification between types.

$$
\begin{aligned}
clauses \in NamedClauses &= (ClauseName \times \mathbb{N}) \rightarrow (\overrightarrow{TypeVariable} \times \overrightarrow{Type}) \\
tdefs \in TypeDefs &= (ConstructorName \times \mathbb{N}) \rightarrow \\
&\qquad (UserDefinedTypeName \times \overrightarrow{TypeVariable} \times \overrightarrow{Type}) \\
\Gamma \in TypeEnv &= Variable \rightarrow Type
\end{aligned}
$$

Figure 8.6: Formalized definitions which are used throughout typechecking.

8.4.3 Helper Functions and Typing Rules

The typing rules rely on a number of helper functions, which have been formally defined in Appendix D. A brief informal description of each helper function follows:

- `keys`: Returns a set of keys in a given map
- `envVariableType`: Given a type environment and a variable, returns the type of the variable along with a new type environment. This signature is non-standard, and reflects the fact that this function really performs one of two operations:

  1. If the variable is already in scope (i.e., in the input type environment), it returns the type of that variable in the type environment, along with the type environment as-is.

2. If the variable is **not** in scope, it adds the variable to the type environment, along with some unknown type. The variable returned is of the same unknown type. Mathematically, this unknown type is treated as a nondeterministic choice between all possible types, though in the implementation this is handled efficiently by employing first-order unification. That is, the unknown type returned is actually a fresh, uninstantiated logical variable in the implementation.

- `typeofTerms`: Given an input type environment and a list of terms, produces a pair holding the types of each term, along with the output type envionment. The type environment is chained along through the terms from the left to the right.

- `typeReplace`: Given a mapping of type variables to types, along with an input type, performs type replacement on the input type yielding an output type. This amounts to a find-and-replace operation of type variables to types.

- `typeReplaceList`: Like `typeReplace`, but it operates over an input list of types and produces an output list of types instead.

As shown in Figure 8.5 there are a number of nested AST components. However, the only components they have in common are x (variables) and i (integer constants), which are uniformly handled. As such, we present all these rules together in the full typing rules shown in Figure 8.7.

While Typed-Prolog is a very different language from SimpleScala (shown in Chapter 7, Section 7.2), the typing rules for the two languages are reminiscent of each other, reflecting the fact that both type systems have roots in Hindley-Milner [178, 179]. That said, there are two key differences between Typed-Prolog and SimpleScala:

1. Higher-order relations in Typed-Prolog operate over a list of types, as opposed

$$\frac{(\tau \cdot \Gamma') = \texttt{envVariableType}(\Gamma, x)}{\Gamma \vdash x : \tau \cdot \Gamma'} \text{ (VAR)} \quad \frac{}{\Gamma \vdash i : \textsf{int} \cdot \Gamma} \text{ (INT)}$$

$$\frac{\Gamma_1 \vdash e_1 : \textsf{int} \cdot \Gamma_2 \quad \Gamma_2 \vdash e_2 : \textsf{int} \cdot \Gamma_3}{\Gamma_1 \vdash e_1 \oplus e_2 : \textsf{int} \cdot \Gamma_3} \text{ (EXP-BINOP)} \quad \frac{\Gamma_1 \vdash e : \textsf{int} \cdot \Gamma_2}{\Gamma_1 \vdash \otimes e : \textsf{int} \cdot \Gamma_2} \text{ (EXP-UNOP)}$$

$$\frac{}{\Gamma \vdash a : \textsf{atom} \cdot \Gamma} \text{ (TERM-ATOM)}$$

$$\frac{(\vec{\tau} \cdot \Gamma_2) = \texttt{typeofTerms}(\Gamma_1, \overrightarrow{term}) \quad \Gamma_2 \vdash body : \textsf{bool} \cdot \Gamma_3}{\Gamma_1 \vdash \textsf{lambda}(\overrightarrow{term}, body) : \textsf{relation}(\vec{\tau}) \cdot \Gamma_1} \text{ (TERM-LAMBDA)}$$

$$\frac{\begin{array}{c} n = |\overrightarrow{term}| \quad (cn \cdot n) \in \texttt{keys}(tdefs) \quad (un \cdot \overrightarrow{T} \cdot \vec{\tau_1}) = tdefs((cn \cdot n)) \\ (\vec{\tau_2} \cdot \Gamma_2) = \texttt{typeofTerms}(\Gamma_1, \overrightarrow{term}) \quad \vec{\tau_3} = \texttt{unifyPoly}(\overrightarrow{T}, \vec{\tau_1}, \vec{\tau_2}) \end{array}}{\Gamma_1 \vdash cn(\overrightarrow{term}) : un(\vec{\tau_3}) \cdot \Gamma_2} \text{ (TERM-CONS)}$$

$$\frac{\Gamma_1 \vdash lhs : \textsf{int} \cdot \Gamma_2 \quad \Gamma_2 \vdash e : \textsf{int} \cdot \Gamma_3}{\Gamma_1 \vdash lhs \ \textsf{is} \ e : \textsf{bool} \cdot \Gamma_3} \text{ (BODY-IS)} \quad \frac{\Gamma_1 \vdash body : \textsf{bool} \cdot \Gamma_2}{\Gamma_1 \vdash \neg body : \textsf{bool} \cdot \Gamma_2} \text{ (BODY-NEG)}$$

$$\frac{\Gamma_1 \vdash body_1 : \textsf{bool} \cdot \Gamma_2 \quad \Gamma_2 \vdash body_2 : \textsf{bool} \cdot \Gamma_3}{\Gamma_1 \vdash body_1 \ bbinary \ body_2 : \textsf{bool} \cdot \Gamma_3} \text{ (BODY-BINOP)}$$

$$\frac{\Gamma_1 \vdash e_1 : \textsf{int} \cdot \Gamma_2 \quad \Gamma_2 \vdash e_2 : \textsf{int} \cdot \Gamma_3}{\Gamma_1 \vdash e_1 \ominus e_2 : \textsf{bool} \cdot \Gamma_3} \text{ (BODY-CMP)}$$

$$\frac{\Gamma_1 \vdash term_1 : \textsf{relation}(\vec{\tau}) \cdot \Gamma_2 \quad (\vec{\tau} \cdot \Gamma_3) = \texttt{typeofTerms}(\Gamma_2, \overrightarrow{term_2})}{\Gamma_1 \vdash \textsf{call}(term_1, \overrightarrow{term_2}) : \textsf{bool} \cdot \Gamma_3} \text{ (BODY-ANONCALL)}$$

$$\frac{\begin{array}{c} n = |\overrightarrow{term}| \quad (cln \cdot n) \in \texttt{keys}(clauses) \quad (\overrightarrow{T} \cdot \vec{\tau_1}) = clauses((cln \cdot n)) \\ (\vec{\tau_2} \cdot \Gamma_2) = \texttt{typeofTerms}(\Gamma_1, \overrightarrow{term}) \quad \vec{tau_3} = \texttt{unifyPoly}(\overrightarrow{T}, \vec{\tau_1}, \vec{\tau_2}) \end{array}}{\Gamma_1 \vdash cln(\overrightarrow{term}) : \textsf{bool} \cdot \Gamma_2} \text{ (BODY-CALL)}$$

Figure 8.7: Typing rules over the syntax defined in Figure 8.5, using the typing domains defined in Figure 8.6

to having distinct input and output types as with higher-order functions in SimpleScala. On a related note, higher-order relations accept an arbitrary number of arguments, whereas the higher-order functions in SimpleScala are restricted to take

only a single input and return only a single output.

2. SimpleScala requires users to annotate the expected values of type variables whenever a polymorphic function is called or a generic constructor is used. In Typed-Prolog, these expected types are inferred with the `unifyPoly` helper function. This partially reflects the fact that SimpleScala was designed to be easier to implement than it was to use, whereas Typed-Prolog was designed to be easy to use with only minimal attention paid to keeping the implementation simple. This also reflects the fact that the implementation language for Typed-Prolog was itself Typed-Prolog, which natively contains first-order unification and logical variables. These features make implementation much easier, as they are more conducive to performing type inference.

8.5 Compiling Higher-Order Relations

While the idiomatic way of abstracting over computation in CLP requires the use of an `eval`-like construct (namely `call`), this construct is not required in order to handle the **call** operation (shown previously in Section 8.4) in Typed-Prolog. In fact, this construct can be translated into normal, portable CLP without the use of any metaprogramming operations. This section discusses exactly how this translation is performed.

8.5.1 Background: Reynolds' Defunctionalization

Reynolds [187] discusses the problem of translating higher-order programs into first-order programs at length, and describes a relatively simple solution known as *defunctionalization*. Defunctionalization concerns exactly how closures, the data structures backing higher-order functions, are created and called. With defunctionalization, syn-

tactic higher-order functions are replaced with first-order code which creates a closure. This closure includes a tag which uniquely identifies the source code to execute when the closure is executed, along with the values of any variables the closure captures. When higher-order functions are called, this tag is used to dispatch on the appropriate source code to execute. In addition, if the closure captured any variables, then these are extracted from the closure during code execution.

To see how this works, an example in C will be used. This example adds in three components not normally in C:

1. The type `INT_TO_INT`, which denotes a function that takes an `int` and returns an `int`

2. The keyword `LAMBDA`, which is used to create a higher-order function. The first parameter to `LAMBDA` is the input variable, and the second parameter is the source code to execute, which may close over variables in the surrounding scope.

3. The keyword `CALL`, which is used to call a higher-order function. The first parameter to `CALL` is the function to call, and the second parameter is the actual parameter to use for the called function.

The resulting C code instrumented with higher-order capabilities is shown in Figure 8.8.

The C code in Figure 8.8 implements a form of the usual functional `map` operation, specialized to operate with both an input and output list of integers. This operation is encoded in the `intToIntMap` function, and called in the `main` function. The `main` function introduces two higher-order functions, one which multiplies its input by `5` (`multByFive`), and another which adds some amount to its input (`addAmount`). Of importance is that `addAmount` closes over a variable in its environment, namely `amt`.

```
1  #include <stdlib.h>
2
3  int* intToIntMap(int* items, size_t len, INT_TO_INT f);
4
5  int* intToIntMap(int* items, size_t len, INT_TO_INT f) {
6    int* retval = (int*)malloc(len * sizeof(int));
7    size_t x;
8    for (x = 0; x < len; x++) {
9      retval[x] = CALL(f, items[x]);
10   }
11   return retval;
12 }
13
14 int main(int argc, char** argv) {
15   int amt = 7;
16   INT_TO_INT multByFive = LAMBDA(int in, return in * 5);
17   INT_TO_INT addAmount = LAMBDA(int in, return in + amt);
18   int input[4] = {0, 1, 2, 3};
19   size_t len = 4;
20   int* list1 = intToIntMap(input, len, multByFive);
21   int* list2 = intToIntMap(input, len, addAmount);
22   free(list1);
23   free(list2);
24   return 0;
25 }
```

Figure 8.8: C code instrumented with higher-order capabilities. While this is not legal C, it can be easily translated to legal C with the help of defunctionalization.

With defunctionalization, the C-like code in Figure 8.8 can be translated into normal C. This translated code is shown in Figure 8.9. The most important change between Figures 8.8 and 8.9 is the addition of the `applyIntToInt` function in the defunctionalized code in Figure 8.9. The `applyIntToInt` function will first observe the input closure's tag and then dispatch on the appropriate code for the closure. Additionally, if the closure in question actually closed over variables, then `applyIntToInt` will cast the data in the closure to the expected value whenever the original code used the variable in question. This can be most easily seen with the translation of `addAmount`, which closed over the

```
1  #include <stdlib.h>
2
3  typedef struct Clo1 { size_t tag; void* data; } CLOSURE1;
4
5  int applyIntToInt(CLOSURE1 clo, int in);
6  int* intToIntMap(int* items, size_t len, CLOSURE1 clo);
7  void printIntList(int* items, size_t len);
8
9  int applyIntToInt(CLOSURE1 clo, int in) {
10   switch (clo.tag) {
11   case 0:
12     return in * 5;
13   case 1:
14     return in + *((int*)(clo.data));
15   }
16 }
17
18 int* intToIntMap(int* items, size_t len, CLOSURE1 clo) {
19   int* retval = (int*)malloc(len * sizeof(int));
20   size_t x;
21   for (x = 0; x < len; x++) {
22     retval[x] = applyIntToInt(clo, items[x]);
23   }
24   return retval;
25 }
26
27 int main(int argc, char** argv) {
28   int amt = 7;
29   CLOSURE1 multByFive = {0, NULL};
30   CLOSURE1 addAmount = {1, &amt};
31   int input[4] = {0, 1, 2, 3};
32   size_t len = 4;
33   int* list1 = intToIntMap(input, len, multByFive);
34   int* list2 = intToIntMap(input, len, addAmount);
35   free(list1);
36   free(list2);
37   return 0;
38 }
```

Figure 8.9: Defunctionalized version of the code shown in Figure 8.8. This is legal C.

`amt` variable in `main` at line 30 in Figure 8.9. While at most only one variable was ever captured in this example, this can be easily scaled to multiple variables by making the `data` field of `struct Closure` (on line 3) point to some composite data structure holding all the variables.

8.5.2 Application of Defunctionalization to Typed-Prolog Compilation

The compiler for Typed-Prolog uses the same basic strategy shown in Section 8.5.1 to translate higher-order relations down to typical first-order Prolog. The only major difference is in how the equivalent of the `applyIntToInt` function in Figure 8.9 is implemented in CLP. To see how this function is implemented, consider the Typed-Prolog code in Figure 8.10, which is a rough port of the C code in Figure 8.8 to Typed-Prolog. This Typed-Prolog code contains higher-order features which need to be translated down into first-order CLP. This translation can be seen in Figure 8.11.

```
1  clausedef(map, [A, B], [list(A), relation([A, B]), list(B)]).
2  map([], _, []).
3  map([A|As], Rel, [B|Bs]) :-
4     call(Rel, A, B),
5     map(As, Rel, Bs).
6
7  clausedef(main, [], []).
8  main :-
9     Amount = 7,
10    Input = [0, 1, 2, 3],
11    map(Input, lambda([I, O], O is I * 5), List1),
12    map(Input, lambda([I, O], O is I + Amount), List2).
```

Figure 8.10: Typed-Prolog code using higher-order capabilities. This is a rough port of the C code in Figure 8.8.

As shown in Figure 8.11, dispatching of higher-order code is performed through the

```
1  call_lambda2(lambda2_0, I, O):-
2    O is I * 5.
3  call_lambda2(lambda2_1(Amount), I, O):-
4    O is I + Amount.
5
6  map([], _, []).
7  map([A|As], Rel, [B|Bs]) :-
8    call_lambda2(Rel, A, B),
9    map(As, Rel, Bs).
10
11 main:-
12   Amount = 7,
13   Input = [0, 1, 2, 3],
14   map(Input, lambda2_0, List1),
15   map(Input, lambda2_1(Amount), List2).
```

Figure 8.11: Translation of the Typed-Prolog code in Figure 8.10 down to first-order CLP. While this code was ultimately produced by the Typed-Prolog compiler, it was edited in order to preserve the names of variables and clauses used in Figure 8.10. Section 8.6 discusses the sort of changes which the compiler originally made to the names.

`call_lambda2` procedure. The 2 in `call_lambda2` refers to the number of parameters the lambda expects. In this case, since only `map` is defined which expects a relation of arity 2, only `call_lambda2` rules have been emitted. There are only two rules for `call_lambda2`, corresponding to the fact that only two lambdas were defined in the original code.

The first argument to `call_lambda2` in Figure 8.11 is that of a closure. Closures are represented with structures (or atoms, if they do not close over any variables), where the name of the structure serves as a unique tag, and the contents of the structure are the variables the closure closes over. In this case there are only two closures in play, namely `lambda2_0` and `lambda2_1`. The `lambda2` prefix indicates that the closures expect 2 arguments when called, and the remaining portion of the name is to ensure uniqueness for different closures. As shown, depending on the input closure, different rules to `call_lambda2` will be executed. Assuming the name of the closure is bound

(which, in practice, should always be true), only one `call_lambda2` rule will ever be executed (i.e., the call to `call_lambda2` will be deterministic).

While theoretically this dispatching could entail a linear search through all the different `call_lambda2` rules to see which applies, this is not expected to be the case for modern engines. Specifically, thanks to the common optimization of clause indexing (e.g., [188]), it is expected that determining which clause to execute is no more expensive than a constant-time hash table lookup.

As an aside, it should be noted that while the keyword **call** is used in the Typed-Prolog code in Figure 8.10, the semantics of **call** bear little resemblance to the `eval`-like behavior of `call` in CLP. The naming of **call** reflects how CLP's `call` is *typically* used in practice, though CLP's `call` is significantly more powerful. This extra power in CLP's `call` is, however, precisely why it is difficult to reason about both from a debugging and a typing standpoint, hence it has been weakened to Typed-Prolog's **call** operator.

8.6 Module System

This section describes the module system in Typed-Prolog. I first discuss the basic usage and capabilities, and then discuss how these modules are compiled to regular CLP.

8.6.1 Using the Module System

Each file must begin with a module definition, and there is exactly one module definition per file. This module definition lists everything exported by the module (i.e., the exported procedures and data in the file). For example, consider the following module definition:

```
module(routines, [sum/2, head/2], [myList, option]).
```

The above definition should be placed into a file named `routines.pl`. This definition states that the procedures `sum/2` (`sum` with arity `2`) and `head/2` (`head` with arity `2`) should be exported. Additionally, the user-defined data types `myList` and `option` are exported (`myList` is used in this example because `list` is treated as a built-in type).

Exported procedures and data types can be imported from other files with the `use_module` directive. For example, to import `sum/2` and `myList` from the `routines.pl` file mentioned prevously, the following `use_module` directive would be used:

```
use_module('routines.pl', [sum/2], [myList]).
```

The above directive assumes that `routines.pl` is in the same directory as the file containing the aforementioned `use_module` directive. Modules in other directories can also be accessed via relative paths (i.e., instead of passing `routines.pl`, a relative path to `routines.pl` would be provided). Blanket imports (e.g., `import java.io.*` in Java) are not permitted.

Unsurprisingly, two distinct modules can have procedures or data types with overlapping names, as long as the overlapping names are not exported. For example, consider two modules `A` and `B`, which both define the `D` datatype. If neither `A` or `B` export `D`, no problems will ever occur involving `D`. Additionally, both `A` and `B` may export `D`, as long as neither module tries to import the other module's `D` definition. However, a problem arises if, say, `A` tries to import `D` from `B`. In this case, two possible definitions of `D` exist, and so compilation will fail. Currently, there is no way to resolve this problem without renaming `D` in one of the modules. Possible future solutions to this problem include scoped imports or import renaming (the latter of which Scala [118] allows), though this is not being actively investigated as this limitation helps keep the module system simple.

As for variables, all variables in Typed-Prolog are local, so no name mangling is necessary. That said, the Typed-Prolog compiler does not preserve the original names in

the program, due to uninteresting low-level details regarding how Typed-Prolog code is parsed.

Cyclic module definitions and uses are not allowed. For example, if module `A` issues a `use_module` directive for module `B`, then compilation will fail if module `B` issues a `use_module` directive for module `A`. That said, it is perfectly fine if the module usage layout forms a directed acyclic graph; the compiler will resolve the dependency order automatically with the usual topological sort [189].

8.6.2 Implementation of the Module System in Typed-Prolog

Ultimately, Typed-Prolog will stitch together all modules into a single output CLP file during compilation. In order to ensure that this stitching will not introduce naming ambiguities, name mangling is performed in a way that guarantees that definitions from different modules will remain distinct. This is done by giving each module a unique identifier n, where $n \in \mathbb{N}$. Each definition (i.e., a procedure or a user-defined datatype) is also given a visibility modifier of `public` if the definition is exported, and `private` if the definition is **not** exported. Putting these two points together, the original name `name` is then mangled to `visMod_n_name`, where `visMod` is either `public` or `private` according to the previous explanation, and `n` is the unique module identifier. This ensures that even if two modules have a definition of `name`, these names will still end up being different in the compiled code.

To see this in action, consider modules `modA` and `modB`, which are in Figures 8.12 and 8.13. These have been compiled together with the Typed-Prolog compiler, resulting in the code in Figure 8.14. The effect of name mangling is immediately apprent in Figure 8.14, where `modA` was given the identifier `0` and `modB` was given the identifier `1`. Exported definitions begin with `public`, and non-exported definitions begin with

`private`. The result of defunctionalization of the higher-order relations (see Section 8.5) can also be seen with the `call_lambda2` and `call_lambda3` procedures.

```
1  module(modA, [foldLeft/4], [myList]).
2
3  datadef(myList, [A], [myCons(A, myList(A)), myNil]).
4
5  clausedef(foldLeft,
6            [A, B],
7            [myList(A), B, relation([B, A, B]), B]).
8  foldLeft(myNil, B, _, B).
9  foldLeft(myCons(A, As), B, Rel, Retval) :-
10   call(Rel, B, A, NewB),
11   foldLeft(As, NewB, Rel, Retval).
```

Figure 8.12: Module A, defined in a file named `modA.pl`. This exports `foldLeft` with arity `4`, as well as the `myList` data type.

8.7 Results and Discussion

The bulk of Typed-Prolog was written over the course of three weeks. This was done in phases, where the first phase consisted of writing a very limited compiler directly in CLP. From there, type annotations were added to this compiler to form a simplistic compiler for Typed-Prolog in Typed-Prolog. Features were then incrementally added using this sort of bootstrapping process until the language stabilized to what it is currently.

Once the language was stabilized, a number of test case generators I had written in CLP were ported to Typed-Prolog. This porting process consisted of:

- Replacing uses of metaprogramming (e.g, CLP's `call`) with higher-order relations
- Adding type annotations to clauses (with `clausedef`), and defining user-defined types (with `datadef`)

```
1  module(modB, [flatMap/3, head/2], [option]).
2
3  use_module('modA.pl', [foldLeft/4], [myList]).
4
5  datadef(option, [A], [some(A), none]).
6
7  clausedef(flatMap, [A, B], [myList(A),
8                              relation([A, myList(B)]),
9                              myList(B)]).
10 flatMap(As, Rel, Bs) :-
11   foldLeft(As,
12            myNil,
13            lambda([Accum, A, NewAccum],
14                   (call(Rel, A, CurBs),
15                    myAppend(Accum, CurBs, NewAccum))),
16            Bs).
17
18 clausedef(myAppend, [A], [myList(A), myList(A), myList(A)]).
19 myAppend(myNil, Lst, Lst).
20 myAppend(myCons(A, As), Lst, myCons(A, Rest)) :-
21   myAppend(As, Lst, Rest).
22
23 clausedef(head, [A], [myList(A), option(A)]).
24 head(myNil, none).
25 head(myCons(A, _), some(A)).
26
27 clausedef(flatMapTest, [], [myList(int)]).
28 flatMapTest(Lst) :-
29   flatMap(myCons(1, myCons(2, myCons(3, myNil))),
30           lambda([X, myCons(MinOne,
31                             myCons(X,
32                               myCons(PlusOne,
33                                 myNil)))],
34                  (MinOne is X - 1,
35                   PlusOne is X + 1)),
36           Lst).
```

Figure 8.13: Module B, defined in a file named `modB.pl`. This exports `flatMap` with arity `3`, `head` with arity `2`, and the `option` data type. From Module A in `modA.pl`, this imports the procedure `foldLeft` with arity `4`, and the `myList` data type.

```
1  call_lambda2(lambda2_1, X, public_0_myCons(MinOne,
2                                 public_0_myCons(X,
3                                   public_0_myCons(PlusOne,
4                                     public_0_myNil)))):-
5    MinOne is X - 1,
6    PlusOne is X + 1.
7
8  call_lambda3(lambda3_0(Rel), Accum, A, NewAccum):-
9    call_lambda2(Rel, A, CurBs),
10   private_1_myAppend(Accum, CurBs, NewAccum).
11
12 public_1_flatMap(As, Rel, Bs):-
13   public_0_foldLeft(As, public_0_myNil, lambda3_0(Rel), Bs).
14
15 private_1_myAppend(public_0_myNil, Lst, Lst).
16 private_1_myAppend(public_0_myCons(A, As), Lst,
17                     public_0_myCons(A, Rest)) :-
18   private_1_myAppend(As, Lst, Rest).
19
20 public_1_head(public_0_myNil, public_1_none).
21 public_1_head(public_0_myCons(A, _), public_1_some(A)).
22
23 private_1_flatMapTest(Lst):-
24   public_1_flatMap(public_0_myCons(1,
25                      public_0_myCons(2,
26                        public_0_myCons(3,
27                          public_0_myNil))),
28                    lambda2_1,
29                    Lst).
30
31 public_0_foldLeft(public_0_myNil, B, _, B).
32 public_0_foldLeft(public_0_myCons(A, As), B, Rel, Retval):-
33   call_lambda3(Rel, B, A, NewB),
34   public_0_foldLeft(As, NewB, Rel, Retval).
```

Figure 8.14: Result of compiling modules `modA` (in Figure 8.12) and `modB` (in Figure 8.13) together. For the sake of clarity, extra spaces were added, and variables were renamed to make them consistent with the variable names in the source files.

- Converting any uses of the CLP module system to Typed-Prolog's module system

This porting process revealed the presence of type errors in each test case generator ported, with multiple type errors in some cases. While some of these type errors were due to the use of dynamic CLP features, others revealed true bugs in the code which unintentionally restricted the sort of tests produced by the test case generator in play.

Since then, I have used Typed-Prolog for every test case generator I have implemented, and my productivity has significantly increased. The most important benefit of Typed-Prolog over plain CLP is that I can refactor code with confidence, which is important in domains where the biggest challenge is in figuring out what kind of tests to generate (as with Rust in Chapter 4).

As for performance gains, by removing metaprogramming from the picture, a 14× speedup was observed on GNU Prolog [66] for a `foldLeft`-heavy program. The same program saw a 3× improvement on SWI-PL [67]. These gains are all the more significant considering that the code structure either did not change or became even simpler.

8.8 Conclusions

Typed-Prolog has led to far more reliable test case generators, and has allowed me to implement and refactor test case generators with confidence. Overall, it has allowed me to focus less on mundane tasks, freeing up mental resources to focus more on the more fundamental generation task at hand. While this work was mostly an engineering effort, my intention is to publish this in tandem with the metainterpreter in Chapter 9. Collectively, these contributions serve to make CLP easier to use for test case generation.

Chapter 9

Improving CLP for Testing: Bounding and Search-Oriented Metainterpreter

9.1 Introduction and Motivation

This chapter discusses a mechanism which can be used to automatically bound the constraint search space and dramatically change CLP's constraint search strategy. Such capabilities are of great use to test case generation, as this allows us to easily experiment with multiple search strategies, along with separating the code related to constraint search from the code dealing with constraints themselves. I have used these capabilities for years in multiple test case generators, as these capabilities help solve common problems which arise in test case generation. While these capabilities do not serve to strengthen my overall thesis, I felt it necessary to discuss these as I have used them extensively over the past few years. With this in mind, this chapter does not provide evidence for CLP's effectiveness for structured black-box test case generation, but rather discusses technical details which are relevant to anyone who wants to use CLP for test case generation.

While Typed-Prolog (discussed in Chapter 8) certainly assists in the test case gen-

eration process, there are still a number of problems which need to be addressed. In particular, this chapter discusses two major remaining problems which are specific to test case generation in CLP:

1. The search strategy of CLP is fixed to be a deterministic depth-first search. This is problematic when we must sample the space of possible test inputs (as when the space is prohibitively large), as the samples tend to be biased.

2. In practice, it is necessary to bound a number of components of generation (as discussed in Chapter 7, Section 7.3). Most of these bounds are simple but tedious to apply, and ultimately pollute generator code with bounding concerns.

With the first problem, it is often the case that the space of possible test inputs is in the billions or even trillions of inputs, even for deceptively small bounds. Such large spaces are particularly common if bounding is depth-based, as the size of the state space grows exponentially with the size of the bound. Such large state spaces make it a practical necessity to sample the state space; that is, generate only a subset of the entire state space. The problem here is that thanks to CLP's deterministic DFS, the samples are highly organized, with an extremely heavy bias towards rules which were placed early in the file. To illustrate this, consider the CLP code in Figure 9.1, which encodes a generator for bounded arithmetic expressions.

Given any arbitrary bound (the first parameter to `arith` in Figure 9.1), this generator will *eventually* produce all programs. However, if we restrict ourselves to picking only the first k programs, this will result in a sample that is biased towards `plus`, as `plus` is the first rule present. For example, with bound `3`, none of the first 10 programs generated even contain `minus`. Additionally, the first program that begins with `minus` only appears when we have searched approximately halfway through the entire search space. Even for the seemingly-small bound of `3`, this means that a whopping 40,807 programs have

```
1  decBound(In, Out) :-
2    In > 0,
3    Out is In - 1.
4
5  arith(_, int(0)).
6  arith(_, int(1)).
7  arith(Bound, plus(E1, E2)) :-
8    decBound(Bound, NewBound),
9    arith(NewBound, E1),
10   arith(NewBound, E2).
11 arith(Bound, minus(E1, E2)) :-
12   decBound(Bound, NewBound),
13   arith(NewBound, E1),
14   arith(NewBound, E2).
```

Figure 9.1: Simple generator for depth-bounded arithmetic expressions, consisting of the integers 0 and 1, with the operations + and −. Note that the + operation (`plus` at line 7) is before the − operation (`minus` at line 11).

to be generated in order to observe **just one** which starts with `minus`. These sort of skewed samples can lead to missed bugs, ultimately because the test case generator is too predictable.

As for the second problem of polluting the code with bounding-related concerns, this can again be seen in Figure 9.1. Most apparent here is the `decBound` procedure defined on lines 1-3, with subsequent calls on lines 8 and 12. While these bounding-related components are intuitively simple to deal with, they bloat the code. An extra bounding-related parameter had to be added to `arith`, and recursive calls to `arith` must be sure to call `decBound`. It is relatively easy to forgot to call `decBound`, which can result in a loop where ever-deeper programs are produced (discussed in Section 7.3.1). Worse yet, if the forgotten `decBound` call is positioned relatively late in the rules, it may take a significant amount of time before this sort of loop is hit. Speaking from experience, this can be hours or even days, leading to a false sense of security that the generator in question is correct. As such, ideally we want to remove `decBound`, the associated `decBound` calls,

and the explicit `Bound` parameter from the code in Figure 9.1 altogether, though still somehow get the benefits of bounding.

In order to solve these problems, we take the approach of implementing a CLP *metainterpreter*; that is, an interpreter for CLP written in CLP. By running the generator code on the metainterpreter, we can subtly change the semantics of CLP itself, but only for the generator code. These semantic changes can impart search strategies other than just CLP's deterministic DFS, solving the first aforementioned problem. These semantic changes can also introduce failure at well-defined bounding points, leading to behavior that is equivalent to `decBound` in Figure 9.1, but without any changes to the generator code. This introduced failure ultimatley solves the second aforementioned problem.

While we *could* design a problem-specific metainterpreter for every test case generator we want to implement, this would be a relatively daunting task. As such, a design goal of our metainterpreter is that the interpreter itself never needs to be modified, and so it is widely applicable to all sorts of test case generators. However, the interpreter is easy to customize for a particular generation problem. This all is made possible by a novel abstraction wherein the search strategy itself is a parameter to the metainterpreter. Different instantiations of this search strategy yield radically different behaviors, all without any changes to the metainterpreter itself.

While a variety of search strategies are possible, we observe that most realistic strategies can be explained as compositions of simpler, more fundamental strategies. A second novel contribution is in how these sort of compositions can be performed, which utilizes a combinator-based approach borrowed from functional programming.

Overall, I make the following contributions in this chapter:

- A CLP metainterpreter which parameterizes search and bounding, discussed in Section 9.4

- How different search and bounding strategies for this metainterpreter can be composed from more fundamental units, discussed in Section 9.5

9.2 Related Work

This related work section is dedicated to flaws in our previous work on using CLP to generate JavaScript programs [70], which was partially discussed in Chapter 2. That work claimed to solve a number of problems which, in hindsight, were not solved completely. This only became apparent after attempting to apply the ideas in that work to much more complex problems; as we pushed the limits of CLP's expressibility, entirely new problems arise. In particular, there are two problems I identify in that work [70] which I seek to address with the metainterpreter in this chapter:

1. The approach prior work used for random search does not scale to complex generation tasks

2. The approach prior work used for bounding leads to boilerplate code, and it also fails to scale to complex generation tasks.

More details follow regarding these problems.

9.2.1 Random Search

The approach used in Dewey et al. [70] to add random search to CLP was based on injecting calls to SWI-PL's [67] `maybe` procedure into code. The `maybe` procedure will fail with some given probability, effectively injecting random failure into generation. The idea here was that random failure would cause different test inputs to be generated each time the generator was run. For the relatively simple problems considered in Dewey et al., this approach seemed adequate.

However, with larger generation problems, the `maybe`-based approach becomes untenable. This occurs for two main reasons. For one, since clauses are tried in order with CLP, clauses present earlier in a given procedure with `maybe` injected tend to be called more frequently. This results in a test suite which is biased towards earlier productions in the clause listing. To better understand this problem, consider the following CLP code snippet:

```
1  foo(...) :-
2    maybe(0.5), ...
3  foo(...) :-
4    maybe(0.5), ...
5  foo(...) :-
6    maybe(0.5), ...
7  foo(...) :-
8      ...
```

As shown, the `foo` procedure consists of three rules, the first three of which each contain a `maybe` with a 50% chance of failure. The last clause, in contrast, contains no `maybe`. However, all else being equal, there is nearly an 88% chance that one of the first three rules will be applied (7/8), biasing against the last rule. This can be alleviated by carefully tuning the probabilities in the `maybe` clauses, but this is hardly straightforward.

The second problem with `maybe`-based search is that it causes clauses to randomly fail, as opposed to choosing clauses in a random order. This becomes problematic in situations where the number of applicable clauses is limited by the problem itself. For example, when writing a generator for a typechecker, it is often the case where we need to generate an expression of some known type. In such a situation, only the clauses which can possibly yield the known type are applicable, and in some cases only a single clause is applicable. With `maybe`-based search, we may randomly fail on this necessary clause. Such random failure is inefficient, as it can lead to fruitless exploration over dead space. Moreover, with repeated random failures we can quickly get to the point where *all*

possible alternatives have been exhausted, leading to a total failure to generate anything. While such a failure to generate is a rare occurrance for simple problems, with complex problems this becomes the **common case**.

For these reasons, `maybe`-based search is completely impractical for highly complex test case generators. We need a better way to introduce randomness into CLP's search, which is one of the problems the metainterpreter excels at.

9.2.2 Bounding

The approach used in prior work to handle bounding was based on the simplistic addition of a depth-based bound on to each potentially infinite procedure. This sort of approach is discussed at length in Section 7.3.1. While this approach works with simpler problems, it imparts quite a bit of code bloat and boilerplate, as it requires modification of every rule in a procedure. Moreover, on complex problems, such a simplistic bound can lead to lots of fruitless search, resulting in generators with a poor generation rate. This latter problem is discussed more completely in Section 9.5.3.

9.3 Background on CLP Metainterpreters

The purpose of this section is to provide basic background for CLP interpreters written in CLP (i.e., CLP *metainterpreters*). This background serves as a foundation on which my novel parameterized search and bounding metainterpreter can be built, which is discussed in Section 9.4.

The basic idea of writing CLP metainterpreters is nothing new. Both Sterling et al. [100] and O'Keefe [174] discuss how metainterpreters can be implemented, along with the advantages these sort of metainterpreters offer. For example, Sterling et al. shows how metainterpreters can be used to assist the debugging process; custom metainterpreters

can easily show which code is called when, and basic debugging concepts like viewable stack traces can be implemented with relative ease.

While the idea of writing an interpreter for CLP in CLP may sound daunting, this is a surprisingly simple task. Most operations can be deferred directly to the engine hosting the metainterpreter, including basic functionality like unification and nondeterminism, as well as any low-level functionality like garbage collection. In fact, for most realistic CLP code, it is only necessary to handle the following operations:

- Conjunction (the "`,`" operator)
- Nondeterministic choice (the "`;`" operator)
- Implication (the "`Cond -> True;False`" operator, where `Cond` is a condition to check, `True` is what to execute if the condition is true, and `False` is what to execute if the condition is false)
- Negation-as-failure (the "`\+`" operator)
- Calling to "built-in" operations, which require interaction with the host CLP engine
- Procedure calls

In addition to needing to handle relatively few operations, the job of writing a CLP metainterpreter is further made easy by the fact that CLP is *homoiconic*. That is, the basic syntax of CLP itself is the same as the syntax for data structures in CLP (specifically terms). This is a property most commonly discussed in reference to Lisp-style languages, though CLP is not in the Lisp family. To better see this homoiconicity, consider the CLP snippet below:

```
foo(X) :-
  bar(X),
  baz(X).
```

Thinking in terms of CLP's term language, `foo(X)`, `bar(X)`, and `baz(X)` look term-like, though the rest of the program may not appear to look this way. However, the CLP parser can handle terms written in infix style, which can be used to explain how `:-` and `,` are handled. Specifically, after parsing, the above snippet is internally represented as follows:

```
:-(foo(X), (,(bar(X), baz(X)))).
```

The entire rule is placed into a structure named `:-` with arity 2. The first parameter to `:-` holds the clause head, namely `foo(X)`. The second parameter to `:-` holds the clause body. The body of the clause consists of a single compound term named `,` with arity 2, where the parameters are `bar(X)` and `baz(X)`, respectively.

Putting all this together, a simple albeit complete metainterpreter is presented in Figure 9.2. An example call to this metainterpreter is shown below:

```
?- interpret(foo(bar)).
```

This would call the unspecified `foo` procedure, executed on the metainterpreter. For now, the intention is that this will have the same behavior as the related query:

```
?- foo(bar).
```

The only difference with the latter query is that the latter query executes directly on the host engine, as opposed to being executed indirectly through the metainterpreter (i.e., the `interpret` procedure).

A tour of the code in Figure 9.2 follows. Built-in operations are handled first starting on line 7, as these take precedence over all other operations. We test to see if the input to `interpret` is actually a built-in operation on line 8. This calls the `builtin` procedure starting on line 1, which lists off a number of operations which we defer to the host engine. Specifically in this case, we defer:

```
1  builtin(true).
2  builtin(=(_, _)).
3  builtin(fail).
4  builtin(writeln(_)).
5  builtin(is(_, _)).
6
7  interpret(Builtin) :-
8    builtin(Builtin),
9    !,
10   call(Builtin).
11 interpret(;(->(If, Then, Else))) :-
12   !,
13   (interpret(If) ->
14       interpret(Then);
15       interpret(Else)).
16 interpret((Q1, Q2)) :-
17   !,
18   interpret(Q1),
19   interpret(Q2).
20 interpret((Q1; Q2)) :-
21   !,
22   (interpret(Q1); interpret(Q2)).
23 interpret(\+ Q) :-
24   !,
25   \+ interpret(Q).
26 interpret(Call) :-
27   clause(Call, Body, _),
28   interpret(Body).
```

Figure 9.2: Basic, but nonetheless complete, CLP metainterpreter implementation. The `interpret` procedure serves as the entry point.

- `true` (line 1), `fail` (line 2), and = (line 3). These can be handled relatively easily in the metainterpreter itself, but defering them reduces the size of the metainterpreter (i.e., we do not need special handling for them if we simply defer).

- `writeln`. Because `writeln` performs IO, ultimately we rely on the host engine. We cannot natively perform IO in the metainterpreter.

- `is`. The behavior of `is` is fairly complex as it must parse a given arithmetic expression, so we simply defer it to the host engine.

Of particular importance with all these built-in operations is that it is expected that none of them will call into any of the code we are executing. Such calls back into the executed code are expected to work correctly, but they will effectively escape the metainterpreter. The intention with built-in operations is that we intentionally escape the metainterpreter, but only for one operation at a time, and only then for operations which are on a predefined whitelist (represented with the `builtin` procedure).

If the input is a builtin operation, the call to `builtin` will succeed on line 8, and then the cut (`!`) will be hit on line 9. This cut will prevent us backtracking to other rules, in particular the rule for calls (line 26); the need for this cut and others will be discussed later on in the context of calls. After the cut on line 9, the builtin operation is called in the host CLP interpreter with the `call` built-in, which has the same behavior as `eval` in JavaScript or Python.

Implications ("`Cond -> True;False`", starting on line 11), conjunction ("`,`", starting on line 16) nondeterministic choice ("`;`", starting on line 20), and negation-as-failure ("`\+`", starting on line 23) are each handled in a similar way: if the input term matches the appropriate respective form, perform a cut to prevent backtracking to other `interpret` clauses, and then implement the desired behavior using the corresponding feature in the host engine. Calls are arguably the most complex feature in this metainterpreter, which start on line 26. Unlike the rest of the operators, calls do not have a particular form. Calls are identified by the fact that the rest of the `interpret` rules did not match on them. This is ultimately why cut was needed in the other rules, as otherwise we could nondeterministically treat forms like `(Q1, Q2)` as conjunctions *or* calls. (As an aside, this could be rewritten without cuts by explicitly ensuring that `Call` on line 26 does **not**

match any of the other forms, but this would be redundant.) On line 27, the `clause` operation is executed on the host CLP engine. The `clause` operation takes the form of a call, unifies the call with the heads of any matching clauses, and nondeterministically returns the bodies of these clauses. This operation is very similar to β reduction in lambda calculus, except in CLP the operation is nondeterministic because multiple clauses may apply. Similarly, `clause` may fail if no clauses apply. Once the body is returned, the metainterpreter interprets the body on line `28` in a recursive fashion.

Notably, while this metainterpreter uses the cut (`!`) operation, it does not support the cut operation in input programs. While the cut operation can be implemented with relative ease with the help of exceptions [190], I intentionally omit it as it will not be used later on in Section 9.4. Specifically, Section 9.4.4 provides more detail why cut is omitted.

9.4 A Metainterpreter for CLP that Parameterizes Search and Bounding

This section discusses exactly how the CLP metainterpreter operates with search and bounding completely parameterized. We start this discussion from the basic metainterpreter definition provided in Figure 9.2.

9.4.1 Adding Bounds

The metainterpreter in Figure 9.2 is very simplistic, and offers no features which are not already provided by the host engine. In this subsection, I add an additional feature: depth-based bounding. While the fully-parameterized metainterpreter we will ultimately derive in this section will not bound things in quite the same way, this serves as a starting

point from which to generalize.

We can add a second parameter to the `interpret` procedure in Figure 9.2 which encodes a depth-based bound. If this bound is ever exceeded, then this should trigger failure. This is similar in spirit to the bounding approach discussed in Chapter 7, Section 7.3.1, except now the bounding logic is contained in the metainterpreter itself. This way, the actual generator source code need not change whatsoever. This instrumented metainterpreter is shown in Figure 9.3.

The source code shown in Figure 9.3 should be relatively straightforward. The `interpret` procedure has been instrumented with an additional `Bound` parameter in the first position. Each time `interpret` is to be called recursively, this bound is decremented by the `decBound` procedure starting on line 7. If the bound is exceeded, then `decBound` will fail, thanks to the conditional check on line 8.

While the code in Figure 9.3 trivially makes any search space it operates over finite, it suffers from a major flaw: it decrements the bound for **every** recursive `interpret` call. This is far too course-grained. For example, programs containing lots of conjunctions (“`,`”) may fail, because conjunctions also decrement the bound. With this in mind, it arguably makes more sense to call `decBound` only at line 35; that is, for calls in user-defined code. This ultimately limits the call stack depth, which makes more sense than limiting the number of CLP operations we can perform.

However, limiting `decBound` calls to user-defined calls is arguably still too course-grained. In practice, the only calls which need to be bounded are those which could be infinitely recursive without the bound. Speaking from experience, most generators only have one or two procedures which fit this description. As such, only these potentially problematic procedures should be bounded. With the current bounding strategy shown in Figure 9.3, *every* call would be bounded, whether or not it could lead to infinite recursion. This prevents many test cases from being generated, and in practice this tends to bias

```
builtin(true).
builtin(fail).
builtin(writeln(_)).
builtin(is(_, _)).
builtin(=(_, _)).

decBound(In, Out) :-
  In > 0,
  Out is In - 1.

interpret(_, Builtin) :-
  builtin(Builtin),
  !,
  call(Builtin).
interpret(Bound, ;(->(If, Then, Else))) :-
  !,
  decBound(Bound, NewBound),
  (interpret(NewBound, If) ->
    interpret(NewBound, Then);
    interpret(NewBound, Else)).
interpret(Bound, (Q1, Q2)) :-
  !,
  decBound(Bound, NewBound),
  interpret(NewBound, Q1),
  interpret(NewBound, Q2).
interpret(Bound, (Q1; Q2)) :-
  !,
  decBound(Bound, NewBound),
  (interpret(NewBound, Q1); interpret(NewBound, Q2)).
interpret(Bound, \+ Q) :-
  !,
  decBound(Bound, NewBound),
  \+ interpret(NewBound, Q).
interpret(Bound, Call) :-
  decBound(Bound, NewBound),
  clause(Call, Body, _),
  interpret(NewBound, Body).
```

Figure 9.3: Metainterpreter instrumented with depth-based bounding, based on the metainterpreter in Figure 9.2.

test case generation towards relatively simple tests. This is because more complex tests generally need more work to be generated, which almost always entails recursion over non-problematic procedures (i.e., procedures which do not need bounding).

With all this in mind, a better bounding strategy is to only decrement the bound when we attempt to call a potentially infinitely recursive procedure. This requires maintaining a list of such procedures, in addition to the bound. We only decrement the bound when encountering such procedures. This much more fine-grained generator is shown in Figure 9.4. An example call to this modified metainterpreter is shown below:

```
?- interpret([exp/1], 2, exp(Exp)).
```

The above query states to decrement the bound for the `exp` procedure with arity `1`, and to provide an initial bound of 2. The actual query to execute is `exp(Exp)`.

Looking at the updated metainterpreter code in Figure 9.4, we now pass an additional parameter `RecList` around in the `interpret` procedure. The `decBound` procedure is called only for calls in user code at line 34. The `decBound` procedure now takes the procedures which should be bounded as well as a name/arity pair of what is currently called at line 7. At line 8, if what is currently called is in `RecList`, then the bound is decreased, which can lead to failure if the bound has been exhausted in line 9. However, if what is currently called is **not** a member of `RecList` (that is, the call should not be bounded), then the bound does not change on line 11.

While the metainterpreter in Figure 9.4 is much more fine-grained, it still suffers from some major probelms. For one, there is only one `Bound` parameter. This is problematic if there are multiple procedures which need to decrement the bound, as all use the same bound. Ideally, each procedure should have its own separate bound. However, arguably the largest problem is that this metainterpreter is applicable only to depth-based bounding, when in practice we may want other kinds of bounding as well. These

```
builtin(true).
builtin(fail).
builtin(writeln(_)).
builtin(is(_, _)).
builtin(=(_, _)).

decBound(RecList, What, In, Out) :-
  member(What, RecList) ->
      (In > 0,
       Out is In - 1);
      (Out = In).

interpret(_, _, Builtin) :-
  builtin(Builtin), !,
  call(Builtin).
interpret(RecList, Bound, ;(->(If, Then, Else))) :-
  !,
  (interpret(RecList, Bound, If) ->
      interpret(RecList, Bound, Then);
      interpret(RecList, Bound, Else)).
interpret(RecList, Bound, (Q1, Q2)) :-
  !,
  interpret(RecList, Bound, Q1),
  interpret(RecList, Bound, Q2).
interpret(RecList, Bound, (Q1; Q2)) :-
  !,
  (interpret(RecList, Bound, Q1);
   interpret(RecList, Bound, Q2)).
interpret(RecList, Bound, \+ Q) :-
  !,
  \+ interpret(RecList, Bound, Q).
interpret(RecList, Bound, Call) :-
  functor(Call, Name, Arity),
  decBound(RecList, Name/Arity, Bound, NewBound),
  clause(Call, Body, _),
  interpret(RecList, NewBound, Body).
```

Figure 9.4: Metainterpreter instrumented to only decrement the bound on calls to infinitely recursive procedures. This is based on the metainterpreter in Figure 9.3.

sort of limitations will be addressed in the next subsection (Section 9.4.2).

9.4.2 Parameterizing Bounding

The limitation regarding the restriction to depth-based bounding observed in Figure 9.4 can be addressed by parameterizing the bounding operation itself. While Figure 9.4 currently makes an explicit call to `decBound`, there is no need for this call to be explicit. Instead, we can treat this as a parameter to the metainterpreter itself. A modified metainterpreter which parameterizes bounding is shown in Figure 9.5. An example call to this modified metainterpreter is shown below:

```
?- interpret(3, decBoundCall([exp/1]), exp(Exp), _).
```

The above call will only permit `3` calls to the `exp` procedure with arity `1`, and will initially call `exp(Exp)`.

While the metainterpreter in Figure 9.5 parameterizes the bounding operation, it still does not address the problem of multiple procedures decrementing the same bound. However, an interesting property of this refactor is that this problem is no longer the burden of the metainterpreter itself, but rather the parameters to the metainterpreter. Instead of passing a single monolithic bounding operation, we will incrementally build up the bounding operation from more primitive bounding operations. This is based on a functional combinator approach, much like that of parser combinators (e.g., [191, 192]).

To better understand how this works, types will be employed in a Scala-like context. To illustrate, the type of a bounding operation is as follows:

```
(Call, Bound) => Bound
```

That is, a bounding operation takes a call and an input bound, and produces an output bound. The type of `Bound` will intentionally be left unspecified for the moment. In

```
builtin(true).
builtin(fail).
builtin(writeln(_)).
builtin(is(_, _)).
builtin(=(_, _)).

decBoundCall(RecList, Call, In, Out) :-
  functor(Call, Name, Arity),
  (member(Name/Arity, RecList) ->
    (In > 0,
     Out is In - 1);
    (Out = In)).

% Interpret: InBound, BoundingOp, ToInterpret, OutBound
interpret(Bound, _, Builtin, Bound) :-
  builtin(Builtin), !,
  call(Builtin).
interpret(Bound1, BoundOp, ;(->(If, Then, Else)), OBound) :-
  !,
  (interpret(Bound1, BoundOp, If, Bound2) ->
    interpret(Bound2, BoundOp, Then, Bound3);
    interpret(Bound3, BoundOp, Else, OBound)).
interpret(Bound1, BoundOp, (Q1, Q2), OBound) :-
  !,
  interpret(Bound1, BoundOp, Q1, Bound2),
  interpret(Bound2, BoundOp, Q2, OBound).
interpret(Bound1, BoundOp, (Q1; Q2), OBound) :-
  !,
  (interpret(Bound1, BoundOp, Q1, OBound);
   interpret(Bound1, BoundOp, Q2, OBound)).
interpret(Bound, BoundOp, \+ Q, Bound) :-
  !,
  \+ interpret(Bound, BoundOp, Q, _).
interpret(Bound1, BoundOp, Call, OBound) :-
  call(BoundOp, Call, Bound1, Bound2),
  clause(Call, Body, _),
  interpret(Bound2, BoundOp, Body, OBound).
```

Figure 9.5: Metainterpreter which parameterizes the bounding operation, which generalizes the metainterpreter shown in Figure 9.4.

the nondeterministic context of CLP, this could also fail, indicating that the bound is exceeded.

The general form of a function that builds a bigger bounding operation from a smaller bounding operation has the following signature:

```
def buildUp[A](around: (Call, A) => A, args: ...):
  (Call, (MyBound, A)) => (MyBound, A)
```

In the above snippet, `around` represents the bounding operation which is being wrapped around. The type variable `A` represents the actual bound which is used for the wrapped-around bounding operation, which is permitted to be anything. The variable(s) `args` represents whatever variables are needed for the bounding operation we are attempting to build up. In practice, this will include information about the particular call we are trying to bound, including its name and arity. As for the return value, this still has the same general form of a bounding operation. However, one can see that it now handles a pair of `MyBound` and `A`. The type `MyBound` is left unspecified, though in practice this is usually an integer.

Code following these signatures can be seen in the CLP code in Figure 9.6, which features new bounding operations. The most basic operation is that of the `unbounded` bounding operation on line 1, which simply forwards along the given bound irrespective of the call it was given. The `decBoundCall2` operation has a signature which is compatible with the `buildUp` signature above. The `decBoundCall2` operation takes the following parameters overall:

1. The operation which it wraps around
2. A name/arity pair correponding to the procedure this is supposed to bound
3. The input call to potentially bound

4. An pair consisting of `decBoundCall2`'s bound (`B1`), along with the bound of whatever it is `decBoundCall2` wraps around (`Rest`)

5. An output pair with the same contents as the input pair

If the input call matches up with the name/arity pair `decBoundCall2` is supposed to bound (on line 5), then the bound for `decBoundCall2` is decremented on line 8. If the bound is exhausted, failure occurs on line 7. A cut is used on line 6 to ensure that we will not go to the other rule for `decBoundCall2` on line 9, which is intended for when the name/arity pair does not line up. Line 11 calls the operator `decBoundCall2` wraps around (`Around`), forwarding the bounding operation if `decBoundCall2` does not apply to the particular call `Call`.

With these new bounding operations in play, we can issue queries like the one in Figure 9.7 to the metainterpreter in Figure 9.5.

```
 1 unbounded(_, Bound, Bound).
 2
 3 decBoundCall2(_, Name/Arity, Call,
 4                 pair(B1, Rest), pair(B2, Rest)) :-
 5   functor(Call, Name, Arity),
 6   !,
 7   B1 > 0,
 8   B2 is B1 - 1.
 9 decBoundCall2(Around, _, Call,
10                 pair(B, Rest1), pair(B, Rest2)) :-
11   call(Around, Call, Rest1, Rest2).
```

Figure 9.6: Modified bounding operations useful for the metainterpreter in Figure 9.5.

The query in Figure 9.7 constructs the following composite bounding operation:

- At most `2` calls are permitted to the `isInteger` procedure with arity `1`
- At most `3` calls are permitted to the `exp` procedure with arity `1`

```
?- interpret(pair(2, pair(3, unit)),
             decBoundCall2(decBoundCall2(unbounded,
                                         exp/1),
                           isInteger/1),
             exp(Exp),
             _).
```

Figure 9.7: Example query to the metainterpreter in Figure 9.5.

- Any other calls are permitted without any bounding restrictions, thanks to the `unbounded` bounding operation

At this point, bounding has been fully parameterized. However, we can generalize things further and parameterize search, as well. This search parameterization is explored in the next section (Section 9.4.3).

9.4.3 Adding Search Parameterization

Now that bounding operations have been parameterized in Figure 9.5, it is a surprisingly straightforward jump to parameterize the search strategy, at least to a certain degree. I will first explain exactly how a different search strategy can be encoded, and then I will discuss its actual parameterization.

Modifying the Search Strategy

Observe the call to `clause` in Figure 9.5 at line 37. As previously stated, `clause` nondeterministically returns `Body`, which is bound to the different possible clause bodies which are in play. However, the order in which bodies will be explored will correspond to the same order which is provided in whatever input file is used, so this will behave much like a normal CLP engine.

It is relatively easy to change this ordering with the exploitation of built-in second-

order procedures like `findall`, which can be used to get all the nondeterministic results of a query as a list. For example, consider the following CLP code snippet, which shows a use of `clause` and `findall` in the query:

```
1  exp(int(0)).
2  exp(int(1)).
3  exp(plus(E1, E2)) :-
4    exp(E1),
5    exp(E2).
6
7  ?- Call = exp(int(_)),
8     findall(pair(Call, Body),
9             clause(Call, Body, _),
10            PairList).
11 % ---ENGINE RESPONSE---
12 Call = exp(int(_)),
13 PairList = [pair(exp(int(0)), true),
14             pair(exp(int(1)), true)].
```

The above snippet will perform each call to `clause` ahead of time, using `Call` as a template. For each call to `clause` which is performed, a `pair` is produced, which holds the result of `Call` *after* `clause` is called, along with the `Body` from `clause`. Each pair is placed into an output list called `PairList`, which is in an order reflecting the nondeterministic call order of `clause`. With the above example, one can see that the original call is unchanged (it remains as `exp(int(_))`), though the first element (corresponding to the call) in each produced `pair` in `PairList` has changed. Each body produced (the second element of each `pair`) corresponds to the `exp` rules handling `int`. These bodies simply contain `true`, indicating that there is nothing left to execute (these are specified as facts in the original file).

Once `PairList` is produced, we can nondeterministically select a body from the list, and then execute the body using the `interpret` procedure from Figure 9.5. A simple example of this follows:

```
1 member(pair(Call, Body), PairList),
2 interpret(Bound2, BoundOp, Body, FinalBound).
```

The snippet above will end up interpreting each body, though in the same order as before. However, we can easily manipulate the list *before* calling `member`, like so:

```
1 random_permutation(PairList, PairList2),
2 member(pair(Call, Body), PairList2),
3 ... % rest of code follows
```

The above snippet uses the `random_permutation` procedure (provided in the standard library of SWI-PL [67]), which will derive a random permutation of some input list. Thanks to `random_permutation`, the body selected by `member` and ultimately executed is no longer predictable. This adds a significant degree of randomness to the underlying search.

It should be noted that the underlying strategy is still that of a depth-first search. However, the order in which children are explored is now random, whereas previously it was strictly left-to-right. While this is clearly restrictive in general, in practice this has been observed to radically change the search behavior, and this ultimately leads to a wide distribution of test inputs when sampling is performed.

Search Strategy as a Parameter

Now that it is understood how to yield different search strategies, I will discuss how to parameterize the search strategy. Using types, the behavior of a search strategy can be summarized as follows:

$$\mathrm{List}[\mathrm{Choice}] \Rightarrow \mathrm{List}[\mathrm{Choice}]$$

That is, given an original list of choices to execute, a search strategy reorders them, producing a new list of choices to execute. Each one of these choices will be executed in order, so as long as the result from the choice operator somehow changes the input list,

this will result in a different search. Additionally, there is nothing in this signature that implies that the resulting list of choices must have the same contents as the input list of choices; this is exploited in Section 9.5.5.

The search strategy could be treated as a wholly separate parameter. However, we can cut down on the number of parameters to the metainterpreter and simultaneously increase its generality if the above type signature and the type signature for bounding are combined in some way. The combination used in the final metainterpreter is as follows, using types:

$$(\mathrm{Call}, \ \mathrm{Bound}) \Rrightarrow (\mathrm{Body}, \ \mathrm{Bound})$$

The intuition behind the above type signature is that given a call to perform as well as an input bound, the combined bounding and search strategy will nondeterministically yield a clause body to execute, as well as a new bound. As the yield is nondeterministic, the "function" may simply fail, as when the bound has been exceeded.

The above type signature is a bit of a jump, as it is slightly more general than what is allowed using only the bounding and search type signatures. In practice, it is common to separately build up bounding and search functions, and then combine them at once into a computation satisfying the above type signature. This will be seen later in Section 9.5.2.

With the above type signature in mind, a complete final metainterpreter is shown in Figure 9.8. This metainterpreter differs from the production metainterpreter I use in only two ways:

1. There are more `builtin` facts in the production metaintepreter (e.g., for unsurprising operations like `var`, `nonvar`, `ground`, etc.)

2. The production metainterpreter uses a slightly different type signature for the combined bounding/search operator. The differences merely provide more information about the particular call being made to the different operations (e.g., they include

direct references to the clause called). These differences are not significant for the purposes of this discussion.

```
builtin(true).
builtin(fail).
builtin(writeln(_)).
builtin(is(_, _)).
builtin(=(_, _)).

interpret(Bound, _, Builtin, Bound) :-
  builtin(Builtin),
  !,
  call(Builtin).
interpret(Bound1, Choose, ;(->(If, ThenDo), Else), OBound) :-
  !,
  (interpret(Bound1, Choose, If, Bound2) ->
    (interpret(Bound2, Choose, ThenDo, OBound));
    (interpret(Bound1, Choose, Else, OBound))).
interpret(Bound1, Choose, (Q1, Q2), OBound) :-
  !,
  interpret(Bound1, Choose, Q1, Bound2),
  interpret(Bound2, Choose, Q2, OBound).
interpret(Bound1, Choose, (Q1; Q2), OBound) :-
  !,
  (interpret(Bound1, Choose, Q1, OBound);
   interpret(Bound1, Choose, Q2, OBound)).
interpret(Bound, Choose, \+ Q, Bound) :-
  !,
  \+ interpret(Bound, Choose, Q, _).
interpret(Bound1, Choose, Call, OBound) :-
  call(Choose, Call, Bound1, Body, Bound2),
  interpret(Bound2, Choose, Body, BodyFinal).
```

Figure 9.8: Complete final metainterpreter. `Choose` holds the combined bounding/search operator.

9.4.4 Why Cut (!) is Omitted

Cut (`!`) has intentionally been omitted from the metainterpreter. While the metainterpreter itself uses cut, it is a requirement that code executed on the metainterpreter be devoid of cut. This restriction is because cut stops lacking meaning when the execution order of clauses is no longer deterministic. The semantics of cut fundamentally assume that clauses will be nondeterministically executed in the same order as they appear in the program (that is, CLP is "deterministically nondeterministic", if you will), but we have intentionally relaxed that restriction with the final metainterpreter in Figure 9.8. This would make cut highly unpredictable, as the cut would occur at the first clause that was chosen (which is potentially random), as opposed to the first clause in the file. Without the ability to predict cut, it seems like a relatively useless operation at best, and a source of horrendously unpredictable behavior at worst. As such, cut was intentionally not implemented in the metainterpreter.

While cut is generally discouraged and leads to problems of its own [100, 174], it is still occasionally missed. Cut is useful when encoding cases which should be mutually exclusive, and avoiding cut means putting in lots of seemingly-redundant checks in these scenarios. Worse yet, certain operations (e.g., `once`, which is used to get only the first solution of a query) cannot be implemented without cut. However, considering that the loss of cut enables full bounding and search parameterization, this restriction was deemed a fair compromise.

9.5 Composing Search and Bounding Strategies

The metainterpreter abstraction introduced in the previous section is incredibly general, and allows for a number of different combined bounding and search strategies to be composed together from more fundamental units. This section further describes how

this sort of composition can be done, along with some of the fundamental building blocks which have been implemented.

9.5.1 Easier Composition

In Figure 9.7, the specified bounding operation to `interpret` featured two different uses of `decBoundCall2`. While this works, specifying the bounding operation in this way is undesirable, as it is relatively easy to make a typo. Additionally, there is a certain mental burden involved with manually writing out these sort of composed bounding operations, ultimately because metaprogramming is involved.

For these reasons, the implementation splits up each bounding operation into two separate procedures, for a bounding operation named "`name`":

1. A procedure named "`nameHelper`" which implements the actual operation. This takes the same role as `decBoundCall2` in Figure 9.6.

2. A procedure named "`name`", which constructs the metaprogamming call to "`nameHelper`". By convention, the first parameters to this procedure are any parameters the actual bounding operation in play takes. The second-to-last parameter is the bounding operation that the operation in play should wrap around. The last parameter is the result of the combination; that is, the constructed metaprogramming call.

With this sort of split in mind, the code in Figure 9.7 would be rewritten to form the code in Figure 9.9. In practice, the split shown in Figure 9.9 is far less error-prone, though strictly speaking this leads to more boilerplate in the code.

```
1 ?- decBoundCall2(exp/1, unbounded, Bound1),
2    decBoundCall2(isInteger/1, Bound1, Bound2),
3    interpret(pair(2, pair(3, unit)),
4              Bound2,
5              exp(Exp),
6              _).
```

Figure 9.9: Rewritten version of the code in Figure 9.7, which uses helpers to construct the metaprogramming call as opposed to writing the metaprogramming call directly.

9.5.2 Combining Bounding and Search Operations Cleanly

The previous section (Section 9.4) briefly mentioned that, in practice, it is common to build separate bounding and search functions, and then combine them into one operation. In the implementation, this is done with the `makeChooseOperator` procedure, which takes the following parameters:

1. An input bounding operation to use
2. An input search operation to use
3. An output combined operator

The output combined operator is a metaprogramming call to `makeChooseOperatorHelper`, where `makeChooseOperatorHelper` does the follwing in order:

1. Calls the bounding operator in play to determine the next bound, or outright fail if the bound is exceeded
2. Creates a list of clause bodies compatible with a given call
3. Calls the search operator in play on the list of clause bodies, yielding a final list of clause bodies

4. Nondeterministically selects the clause body to execute, in the order provided by the search operator

Strictly speaking, the production metainterpreter I use does not perform the last step above. In the production metainterpreter the search operator is actually repsonsible for picking which clause body to execute, though there is no practical difference between these two behaviors.

With `makeChooseOperator` in mind, we can perform another rewriting, this time of the code in Figure 9.9 to the code in Figure 9.10. Crucially, this second rewrite is compatible with the final metainterpreter shown in Figure 9.8.

```
?- decBoundCall2(exp/1, unbounded, Bound1),
   decBoundCall2(isInteger/1, Bound1, Bound2),
   makeChooseOperator(Bound2, random_permutation, Choose),
   interpret(pair(2, pair(3, unit)),
             Choose,
             exp(Exp),
             _).
```

Figure 9.10: Rewritten version of the code in Figure 9.9 so it uses `makeChooseOperator` and random search (with the help of `random_permutation`). This code is compatible with the final metainterpreter implementation shown in Figure 9.8.

9.5.3 Intelligently Bounding the Number of Calls

The `decBoundCall2` procedure in Figure 9.6 was used to bound the number of calls to some procedure. The same functionality of `decBoundCall2` is present in the production metainterpreter with the more reasonably-named `boundAtCall` procedure.

While `boundAtCall` may seem like a common fundamental building block, it is rarely used in practice. The reason why is because `boundAtCall` can lead to lots of inefficient search through dead space, particularly when dealing with grammar-based problems

```
1 exp(int(0)).
2 exp(plus(E1, E2)) :-
3   exp(E1),
4   exp(E2).
```

Figure 9.11: Simplistic expression generator

(which, in practice, are most problems). To see why, consider the simplistic expression generator in Figure 9.11. During generation with Figure 9.11, we may begin to generate a tree containing a number of `plus` nodes with the following structure:

```
plus(_, plus(_, plus(_, _)))
```

In order to fill in the holes (`_`) above, we will need at least four calls to `exp`, one for each hole. If we choose to fill any hole with `plus`, then more `exp` calls will be needed.

The problem here is that if we have a bound of `3` on the number of calls to `exp`, then the most we could ever produce is the above incomplete tree. Upon trying to fill any of these holes, we would hit the bound on `exp`, as we used up the entire bound just making the incomplete tree (the tree above contains 3 `plus` nodes, so just producing this requires 3 calls to `exp`). This would trigger failure. However, in most practical cases the next choice to make will have the same problem, again leading to failure. As such, we would end up exploring a large state space that cannot possibly be filled in, as filling in the space completely requires a larger `exp` bound than is remaining.

To solve this problem, a modified variant of `boundAtCall` is employed, named `boundAtCallWithBaseCases`. Like `boundAtCall`, `boundAtCallWithBaseCases` takes a bound on the number of calls permitted to some user-defined procedure. However, `boundAtCallWithBaseCases` also takes a series of base cases for the procedure in play. When the bound is exceeded, `boundAtCallWithBaseCases` will use one of these base cases, as opposed to outright failing. With the incomplete tree example above, this

would lead to a complete tree being emitted, where each hole would be filled with the base case of `int(0)`. This leads to orders of magnitude better performance in practice, with the only downside being that the bound is treated as more of a suggestion than a hard constraint (a similar problem to the one described in Chapter 7, Section 7.4.2). Given that the bounds are relatively arbitrary already (i.e., we just need *some* number in order to keep things finite), this is deemed acceptable.

9.5.4 Dynamically Replacing Calls

Occasionally it is desirable to have a different semantics for procedures when they are being run by the metainterpreter, as opposed to when they are run directly on a CLP engine. For example, consider again the simplistic arithmetic expression generator in Figure 9.11. As written, the only integer constant ever accepted by this generator is `0`. While this may be suitable for a generator, this is likely unacceptable for an acceptor of arithmetic expressions, which generally are assumed to contain arbitrary integer constants. In particular, this can make testing of the generator difficult if we try to test it with known arithmetic expressions. If a known arithmetic expression is rejected, it could either mean there is a bug in the generator, or that simply the `0` restriction came into play. (While this is relatively simplistic in this example, with larger generators this becomes a difficult problem.)

To resolve this problem, one strategy is to rewrite the simplistic generator as shown in Figure 9.12. Figure 9.12 calls out to the `intConstant` procedure for determining if some input integer constant is valid, instead of hard-coding `0` as is done in Figure 9.11. Because `intConstant` is true for every possible input, as written this will accept any possible integer constant. This behavior is desirable when `exp` is used as an acceptor, though not when `exp` is used as a generator, as the generated programs will contain holes

(an issue discussed more thoroughly in Section 7.3.2).

```
1 intConstant(_).
2
3 exp(int(Int)) :-
4   intConstant(Int).
5 exp(plus(E1, E2)) :-
6   exp(E1),
7   exp(E2).
```

Figure 9.12: Refactored generation code from Figure 9.11, modified to handle arbitrary integer constants when used as an acceptor.

It turns out that this problem of holes being emitted when used as a generator can be relatively easily fixed via a custom bounding operation. The intuition behind such an operator follows, specifically in the context of `intConstant` in Figure 9.12. A bounding operation can detect the exact point when a call to `intConstant` is made, as the call is a parameter to the bounding operation. Crucially, the call has not yet been made. At this point, the bounding operation can call *some other procedure*, and then fill in the original call's parameters with the results from the actually-called procedure. For example, the other procedure in this case can produce `0` as a result, and then fill in the result of `intConstant` with `0`. From there, execution proceeds as normal. In this way, we can effectively change the behavior of `intConstant` to fill in `0`, but only if it is run underneath the metainterpreter.

In the implementation, this operation handled by `interceptAndRedirectAtBounding`. This is used fairly often, specifically for the purpose of filling in holes. This also has a nice side benefit of separating generation code from bounding code, as the *other procedure* mentioned above can be located far from the definition of `exp` in Figure 9.12.

9.5.5 Swarm Testing

The last fundamental operation discussed is a custom search operator, as opposed to a custom bounding operator. This can be composed with random search (through `random_permutation`), as well as any other search strategies. While it has the signature of a search operator, it behaves in a very different way from other search operators: instead of reordering choices, it methodically cuts out certain choices.

The operator referred to is that of `addSwarm`, which is based on the idea of Swarm Testing [105]. In Swarm Testing, the idea is to intentionally prevent some subset of AST productions from being emitted. While this reduces the breadth of a test suite generated (they contain fewer productions), it increases the depth of such a suite, as the few productions allowed get used much more thoroughly. In practice, this can lead to more effective bug-finding.

The `addSwarm` operator is used to augment a user-defined procedure with Swarm-like capabilities. The basic idea is that ahead of time, we mark certain rules of a user-defined procedure as being forbidden. When the search operator is called by the metainterpreter, it will prune away calls to any rules which were marked as forbidden. This serves as a simple but effective way of implementing Swarm Testing. Additionally, this generalizes the idea of Swarm Testing to whole procedures, as opposed to just AST productions as in the original formulation [105].

9.6 Results and Discussion

The metainterpreter has been used for multiple years in my work, and it has proven itself indispensible. This has made simple random search a reality, and in most generators the burden of thinking about bounds has been lifted. With bounding, only incredibly complex generators need special thinking (e.g., the SimpleScala generator discussed in

Chapter 7). Additionally, the metainterpreter composes well with Typed-Prolog (discussed in Chapter 8), with Typed-Prolog handling compile-time issues and the metainterpreter handling runtime issues. As such, I consider the metainterpreter to overall be a great success.

The downside of the metainterpreter is that it significantly hurts performance, often by nearly an order of magnitude. While this certainly is significant, in practice it is still usually the case that the test case generation rate exceeds the rate at which tests can actually be executed. Moreover, considering that CLP-based testing already tends to be orders of magnitude faster than existing test case generation tools (see Chapters 3 and 4), even with this significant performance hit the result is still better than the competition.

9.7 Conclusions and Future Work

The metainterpreter has made writing test case generation code simpler, and has allowed for the easy and automatic extension of CLP to proper random testing. With the help of the metainterpreter, I can easily fine-tune bounding parameters without fear of accidentally breaking a generator. Since its development, I have used the metainterpreter extensively in nearly all my test case generation projects.

For future work, a major improvement relates to performance. Instead of working on writing a faster metainterpreter (which gets dangerously close to writing a custom CLP implementation), my plan is to embed the work done by the metainterpreter into a special compiler. This would necessarily restrict the sort of bounding and search operations which could be constructed with the metainterpreter, but based on my experiences with the metainterpreter all commonly-used behavior can be handled. We already have ideas as to how this can be compiled, particularly in the context of a language like Typed-Prolog (discussed in Chapter 8). Such a compilation pass should see a major

performance recovery, and thus make generator execution up to 10× faster even with automatic bounding and search parameterization.

Chapter 10

Conclusions and Future Work

Software bugs are pervasive, necessitating automated techniques in order to find them before they cause problems. In this work, we have looked specifically at the technique of black-box fuzzing for this purpose. After modeling black-box fuzzing as a constraint satisfaction problem, the use of CLP for testing becomes a natural fit given the state of the art in constraint solvers. Through six different case students in disparate domains, I have shown that CLP is widely applicable to testing, and it is overall an effective solution to the generalized structured black-box test case generation problem. During this process, I have discovered and reported dozens of bugs in popular industry-strength software projects, immediately leading to the wider impact of more reliable software. I have also shown how to better adapt CLP to the problem of test case generation.

In the future, I want to apply CLP to even more testing domains in order to further explore CLP's expressibility limits and wider generality. Additionally, I want to experiment with alternative constraint solvers in different testing domains, including the constraint solvers discussed in Chapter 1, Section 1.6. While CLP may represent the pinnacle of expressibility, I am curious to see how much of this expressibility is actually required in simpler testing domains, particularly those in education.

Appendix A

CLP Preliminaries

The purpose of this appendix is to discuss some basic information behind CLP, including a simple introduction. While this introduction is far from complete, it should be sufficient to understand the CLP code used throughout this document. For a more complete discussion of the pragmatics behind writing CLP/Prolog code, books such as *The Art of Prolog* [100] and *The Craft of Prolog* [174] serve as good references. From more of an academic side, Warren et al. [60] is a good reference on how Prolog works in comparison to functional languages, and Jaffar et al. [40, 41] provides the basic theoretical formulation of CLP.

A.1 CLP Background

CLP was first formulated by Jaffar et al. [40]. For our purposes, CLP can be seen as a generalization of Prolog [61], where the domain of discourse is parameterized. For example, the domain of discourse for Prolog is fixed over integers, atoms, and compound structures, whereas this domain of discourse can theoretically be anything for CLP. In practice, this usually includes the same domain of discourse as Prolog, but also includes

features like symbolic integers [146] or sets [193]. From a constraint solver perspective, this relationship between Prolog and CLP is somewhat analogous to the relationhip between SAT and SMT, where SMT can incorporate many types of constraints [57, 45], including those of SAT.

A.2 A Taste of CLP

A.2.1 Overview

CLP is particularly well-suited to problems which involve *constraint satisfaction*, or *search*. In these sort of problems, we know *what* a correct solution looks like, but we may not know exactly *how* to produce it. This is classically true for NP-Complete problems, though this applies to a surprisingly wide context.

The rest of this appendix assumes you are using SWI-PL (`http://www.swi-prolog.org/`) as your *engine*, where the engine is what is used to execute CLP code. SWI-PL can be loaded with the `swipl` command, which will bring you to the REPL. The prompt for the REPL is `?-`; any code you see in this dissertation which is preceeded by `?-` is intended to be written at the REPL. The code after `?-` is known as a *query*, which serves as a code entry point.

CLP code can be loaded in the REPL like so, assuming we are intending to load the code in `foo.pl`:

```
?- [foo].
```

Note that input files **must** have the `.pl` extension.

A.2.2 Facts

One of the most basic things we can do in CLP is define *facts*: statements which are trivially true. For example, consider the following code:

```
1 isInteger(0).
2 isInteger(1).
3 isInteger(2).
4
5 isName(alice).
6 isName(bob).
```

The way to read the above code on a per-line basis is the following:

- 1: `isInteger` is true underneath input `0`
- 2: `isInteger` is true underneath input `1`
- 3: `isInteger` is true underneath input `2`
- 5: `isName` is true underneath input `alice`
- 6: `isName` is true underneath input `bob`

We can issue queries on the above facts like so (where lines which do not begin with `?-` are the output of the engine).

```
1 ?- isInteger(0).
2 true.
3 ?- isInteger(3).
4 false.
5 ?- isName(alice).
6 true.
7 ?- isName(bob).
8 true.
9 ?- isName(carol).
10 false.
```

As shown above, while it is `true` that `isInteger` holds under input `0` in line 1, it is `false` that `isInteger` holds under input `3` in line 3. This follows from the definition of the `isInteger` facts in the previous listing. Similar behavior is seen for the `isName` facts.

Data Used: Integers and Atoms

There are two kinds of data used in the previous listings: integers and atoms. Integers are, well, integers, and are represented as we would expected (e.g., `0`, `1`, `2`, etc.). Atoms are names which begin with a lowercase letter, as with `alice`, `bob`, and `carol` in the above listings. For our purposes, atoms behave like strings, with the constraint that they **must** begin with a lowercase letter. (Strictly speaking, atoms are really *symbols* as opposed to strings, but this distinction will never be important throughout this dissertation.)

A.2.3 Variables

So far, we have only seen how to ask if a fact holds under some known data. In and of itself, this is not particularly useful. However, we can use the same infrastructure to ask a closely related question: under what data does a fact hold?

In order to ask this sort of question, we need to introduce *variables*. Variables **must** begin with an uppercase letter, which distinguishes them from the atoms defined in Section A.2.2. To see variables in action, consider the following code:

```
1 isOne(1).
2 isTwo(2).
```

As before, we can still check to see if a fact holds under some known input:

```
1 ?- isOne(1).
2 true.
3 ?- isOne(2).
4 false.
```

However, with variables in play, we can now ask for inputs under which the fact holds, like so:

```
1 ?- isOne(X).
2 X = 1.
3 ?- isTwo(Y).
4 Y = 2.
```

Variables can also be used in the very definition of facts. For example, consider the following code:

```
areEqual(X, X).
```

Intuitively, the meaning of the above code is that the `areEqual` fact holds if both of its inputs have the same value. When used with known data, this behaves in a straightforward manner:

```
1 ?- areEqual(1, 1).
2 true.
3 ?- areEqual(1, 2).
4 false.
5 ?- areEqual(2, 2).
6 true.
```

However, with variables in play, the `areEqual` fact begins to show some interesting behavior. Consider the following queries:

```
1 ?- areEqual(X, 1).
2 X = 1.
3 ?- areEqual(1, X).
4 X = 1.
5 ?- areEqual(X, Y).
6 X = Y.
```

All three queries above *succeeded*, meaning they were true. However, because the values of some variables were determined in the process, the engine no longer prints `true`; the `true` is implicit in the fact that some variables received values. At line 1, the engine figured out that in order to make the `areEqual` fact hold underneath the variable `X` and

the integer `1`, it must be the case that `X = 1`. This was similarly true at line 3, where the only difference is the order of the parameters to `areEqual`. The query on line 5 is arguably the most interesting of these, because the engine was able to deduce `X = Y`; that is, the variables `X` and `Y` must be equal to each other. The engine deduced this without knowing exactly what these variables hold; while we know that `X = Y`, it could be the case that `X = 1` or `X = 2`, but we do not know for certain.

Let's go deeper. Consider the code below, which behaves a lot like `areEqual` above but it operates over three inputs instead of two:

```
tripleEqual(X, X, X).
```

Now consider the following queries:

```
 1 ?- tripleEqual(1, 1, 2).
 2 false.
 3 ?- tripleEqual(X, 1, 2).
 4 false.
 5 ?- tripleEqual(1, 1, X).
 6 X = 1.
 7 ?- tripleEqual(X, Y, 1).
 8 X = Y, Y = 1.
 9 ?- tripleEqual(X, 1, Y).
10 X = Y, Y = 1.
11 ?- tripleEqual(X, Y, Z).
12 X = Y, Y = Z.
```

As one might expect, the query on line 1 does not succeed (it gives back `false`), because `1` and `2` are not equal to each other. The same reasoning applies for the query on line 3; the value of variable `X` is irrelevant, because `1` will never equal `2`. The query on line 5 succeeds with `X = 1`, as all other inputs have been fixed at `1`. The query on line 7 shows that all the variables involved are equal to each other (namely, `X = Y`), but additionally one of those variables equals a value (namely, `Y = 1`). While the engine does not explicitly show it, it transitively holds that `X = 1` for this query, as `X = Y` and `Y = 1`. The exact same reasoning follows for the query on line 9, which differs from line 7 only in parameter

ordering. With the query on line 11, all variables are equal to each other (again, while the engine does not explicitly show it, it transitively holds that `X = Z`, as `X = Y` and `Y = Z`).

A.2.4 Nondeterminism

An astute reader may have noticed that in the previous section (Section A.2.3), we did not use the same initial example introduced in Section A.2.2. For the purposes of the current section, we will reuse the initial example from Section A.2.2, and use it along with the variables introduced in Section A.2.3. For convenience, this code has been duplicated below:

```
1  isInteger(0).
2  isInteger(1).
3  isInteger(2).
4
5  isName(alice).
6  isName(bob).
```

Consider the following query on the above code, which uses variables:

```
?- isInteger(X).
X = 0
```

The engine appears to hang at this point, as it is waiting for user input. If we hit semicolon (`;`), we can see additional output:

```
?- isInteger(X).
X = 0 ;
X = 1
```

...at which point the engine appears to hang again. After another press of semicolon, we receive more output:

```
?- isInteger(X).
X = 0 ;
X = 1 ;
X = 2.
```

...at which point the prompt returns and the engine waits for another query.

This sort of behavior may at first appear very strange, as this illustrates a fundamental feature of logic programming languages which is not present in other types of languages. This feature is that of *nondeterministic execution*. What this means is that execution can split-off into effectively different worlds at well-defined points. In the query above, when executing `isInteger(X)`, there are three possibilities based on the `isInteger` facts. As such, execution splits into three different worlds, where each world executes a different `isInteger` fact. A description of each world follows:

- In the first world, the fact `isInteger(0)` is chosen, so `X = 0`.
- In the second world, the fact `isInteger(1)` is chosen, so `X = 1`
- In the third world, the fact `isInteger(2)` is chosen, so `X = 2`

Note that while execution split into three worlds, we visited each of these three worlds in a precise order which reflected the order of the rules in the file. Each time we pressed semicolon, we effectively requested that the engine explore the next world.

Each separate use of `isInteger` in a query (hereinafter called a *call*) splits the world in this way. That is, if multiple calls to `isInteger` are made, then *each* will split execution in this manner. An example follows.

Conjunction

We can make multiple calls to `isInteger` via *conjunction*, represented with a comma (`,`). This is also sometimes referred to as a *compound query*. To see this in action, consider the following query:

```
1 ?- isInteger(X), isInteger(Y).
```

Intuitively, because each call to `isInteger` splits the world, we would expect this query to show all possible combinations of `X` and `Y` if we keep hitting semicolon for more solutions. While this will happen, these solutions will be in a well-defined order. Because there are so many solutions, we put these in a separate listing below for clarity, in the same order as delivered by the engine:

1. `X = 0, Y = 0`
2. `X = 0, Y = 1`
3. `X = 0, Y = 2`
4. `X = 1, Y = 0`
5. `X = 1, Y = 1`
6. `X = 1, Y = 2`
7. `X = 2, Y = 0`
8. `X = 2, Y = 1`
9. `X = 2, Y = 2`

As shown with the query results above, the world splits on the first call to `isInteger(X)`. In the first world explored, the fact `isInteger(0)` is chosen, and so `X = 0`. Execution then proceeds forward with `X = 0`, until the second call to `isInteger(Y)` is encountered. This call selects the `isInteger(0)` fact to use, so `Y = 0`. At this point, computation has ended for this set of worlds, leading to the overall result `X = 0, Y = 0`.

However, there are still other worlds to explore. The engine will explore different worlds based on the most recent world choice made. In this case, because the most

recent choice of worlds was done with `isInteger(Y)`, we choose a different world for this call. The next world has the fact `isInteger(1)`, so `Y = 1`. Nothing is left to execute, so overall `X = 0, Y = 1`. Upon asking for another solution, there are still worlds to explore for the call `isInteger(Y)`, and the fact `isInteger(2)` is chosen, leading to `Y = 2`. Since there is nothing left to compute after this point, overall `X = 0, Y = 2`.

At this point, there are no further choices for `isInteger(Y)`. As such, when asked for another solution, the engine looks for the next most recent choice made, which was done for `isInteger(X)`. Initially, the fact `isInteger(0)` was used for the call to `isInteger(X)`, but at this point we have completey exhausted all the possibilities of that world. As such, the next choice for `isInteger(X)` is made, leading to the usage of the fact `isInteger(1)`, leading to `X = 1`. From here, the call to `isInteger(Y)` is performed, which entails splitting into three worlds corresponding to the three facts for `isInteger`.

In order to implement this idea of going back to the most recent choice, we can employ a stack. Note that this stack maintains *choices*: other worlds which we have yet to explore, with the next world to explore on top. This is related to the call stack only in that both are stacks; the choice stack and the call stack are completely separate data structures.

Revisiting Facts

The original explanation of facts in Section A.2.2 intentionally didn't mention nondeterminism in order to simplify the discussion. However, even for the queries where the input data was known (e.g., `isInteger(1)`), nondeterminism was present, although hidden. To see how this nondeterminism was hidden, consider the query below:

```
1 ?- isInteger(2).
2 true.
```

The above query calls `isInteger`, and it will cause the world to split. In the first world chosen, the fact `isInteger(0)` is used. However, this fact does not work with the given input: we are asking if `isInteger(2)` is true, but instead we are given that `isInteger(1)` is true. These incompatible facts trigger *failure*, the opposite of success. However, this does not completely stop execution, because we see that we have other choices to explore. As such, we then consider the next world: the fact `isInteger(1)`. This similarly leads to failure, but again there are still worlds to explore. Specifically, we still need to explore the fact `isInteger(2)`, which is the next (and last) world. Upon choosing this fact, the original query of `isInteger(2)` succeeds: this query is compatible with a world wherein `isInteger(2)` is a fact.

As described, failure internally occurs twice in execution of this seemingly simple query, but the query still overall succeeded. The query succeeds as long as **any single** choice works; it is ok (and common) if some choices lead to failure. In this way, failure should not be treated like an error condition, but instead a normal part of computation for CLP. Failure exists because we don't generally know ahead of time which choice will work, so we may need to explore choices which ultimately don't work out.

Only when **all** choices lead to failure does a query fail as a whole. To illustrate this, consider the query below:

```
1 ?- isInteger(3).
2 false.
```

Internally, this query follows the exact same pattern as previously described. However, since there is no `isInteger(3)` fact, this will eventually fail. In the process, the facts `isInteger(0)`, `isInteger(1)`, and `isInteger(2)` are all explored in their respective worlds, but none of them lead to success.

As an aside, while this whole discussion is true in general, most engines (SWI-PL included) will implement optimizations (such as *clause indexing*, e.g., [188]) which try to

avoid exploring choices which will lead to failure. For example, consider the following query:

```
1 ?- isInteger(0).
2 true.
```

According to this discussion, the above query should hang, waiting for a semicolon in order to explore additional solutions. However, this will not happen. This is because the engine knows that the other choices for the `isInteger` fact will not work in this case, since there is only one `isInteger(0)` fact. As such, it won't even give you the option to try to explore the other `isInteger` facts: the engine knows they won't work, so it has cut those choices out entirely. Because this is ultimately just an optimization, it is not discussed in detail.

A.2.5 Arithmetic

Arithmetic can be done in CLP using the built-in `is` keyword. Several basic examples follow:

```
1 ?- W is 4 + 6.
2 W = 10.
3 ?- X is 3 * 3.
4 X = 9.
5 ?- Y is 4 / 2.
6 Y = 2.
7 ?- Z is 10 - 5.
8 X = 5.
```

Expressions can be nested, as one might expect:

```
1 ?- W is 2 * (1 + 1).
2 W = 4.
```

Variables can be used in expressions, as long as they have known values. For example, the following is ok:

```
1 ?- X is 2 + 2, Y is X + X.
2 X = 4, Y = 8.
```

...however, the following closely related query is not ok:

```
1 ?- Y is X + X, X is 2 + 2.
2 ERROR: is/2: Arguments are not sufficiently instantiated
```

The above query doesn't work because conjunction (`,`) works **strictly** from left-to-right. With this in mind, `Y is X + X` is executed first. This is problematic, as `X` does not have a value until `X is 2 + 2` is executed. This explains the error message, which states that an argument to `is` does not have a known value (specifically `X` in the above example).

We can preserve the meaning of the above code without being sensitive to the ordering of conjuncts by using built-in arithmetic constraint solvers. Instead of using `is`, we can use the `#=` operator instead, like so:

```
1 ?- use_module(library(clpfd)).
2 % ---FULL OUTPUT OMITTED---
3 true.
4 ?- Y #= X + X, X #= 2 + 2.
5 Y = 8, X = 4.
```

The `use_module(library(clpfd))` directive will tell specifically SWI-PL [67] to load in the constraint solver; different engines perform this in different ways. The particularly library loaded, namely `clpfd`, is short for "constraint logic programming - finite domains"; more details are provided in Triska [146]. The percent symbol (`%`) begins an end-of-line comment. The `#=` operation will perform the given arithmetic operations *symbolically*, which often entails the use of an arithmetic constraint solver. This constraint solving is mostly hidden away from the user. In practice, because such constraint solvers usually come with a performance penalty (up to 10× in my experiences), it is ideal to avoid them until absolutely necessary. It is often possible to achieve this by carefully reordering conjunctions and minor code restructuring.

Related to arithmetic operations are arithmetic comparisons. Consider the following examples:

```
1  ?- 1 < 2.
2  true.
3  ?- 1 =< 1.
4  true.
5  ?- 2 > 1.
6  true.
7  ?- 2 < 1.
8  false.
9  ?- X is 1 + 1, X < 4.
10 true.
```

As with the arithmetic operations, these arithmetic comparisons only work over known values. For example, if we switch the order of the last query above, an error will result. This is shown below:

```
1  ?- X < 4, X is 1 + 1.
2  ERROR: </2: Arguments are not sufficiently instantiated
```

In the above example, since conjunction (`,`) works strictly left-to-right, the `X < 4` portion is executed before the `X is 1 + 1` part. Since the variable `X` has no known value at `X < 4`, we get the error message above.

Once again, we can solve this problem of needing to know the values in relational comparisons with the help of arithmetic constraint solvers. In this case, instead of using `<`, we can use `#<`, like so:

```
1  ?- use_module(library(clpfd)).
2  % ---FULL OUTPUT OMITTED---
3  true.
4  ?- X #< 4, X #= 1 + 1.
5  X = 2.
```

The `use_module(library(clpfd))` directive serves the same purpose as before. Similarly, with `#<`, the `<` operation is now executed symbolically, making the conjunction order irrelevant. This still can come with a significant performance cost, so this sort of change is typically avoided if possible.

A.2.6 Rules

As of yet, we haven't really gotten into how to do much real "work" as far as computation goes. The first example we will use which does real work is that of the factorial function (!) from mathematics. The factorial function can be expressed recursively via the piecewise function shown below:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

The CLP code below implements the factorial function above.

```
1 factorial(0, 1).
2 factorial(N, Result) :-
3   N > 0,
4   MinOne is N - 1,
5   factorial(MinOne, RestResult),
6   Result is N * RestResult.
```

As shown above, we can execute arbitrary code along with a fact by following the fact with `:-`. This symbol is used because it is reminiscent of reverse implication ($\Leftarrow$), which is how calls work from a logical perspective; more details are available in Nilsson et al. [69]. When `:-` is used, it is referred to as a *rule*. Rules are permitted to be recursive, allowing us to perform arbitrary computation. A line-by-line explanation of the above code follows:

1. Fact implementing the behavior that the factorial of 0 is 1, directly from the original math
2. Rule implementing the recursive case of factorial.
3. Check that the input n is greater than 0
4. Calculation of $n - 1$ in the original math

5. Calculation of $(n-1)!$ in the original math

6. Calculation of $n \times (n-1)!$ in the original math

Example queries to the above code are shown below. Note that the engine had additional choices to explore and appeared to "hang" as it did in Section A.2.4. In this case, extra exploration always leads to failure. As such, instead of pressing semicolon for more solutions, period (`.`) was instead pressed to stop the search.

```
 1 ?- factorial(0, N).
 2 N = 1.
 3 ?- factorial(1, N).
 4 N = 1.
 5 ?- factorial(2, N).
 6 N = 2.
 7 ?- factorial(3, N).
 8 N = 6.
 9 ?- factorial(4, N).
10 N = 24.
```

As shown with the example queries above, the first parameter to `factorial` is the number to get the factorial of. The second parameter holds the result. This works very differently from what one might expect from functional programming; instead of there being very well-defined inputs and outputs (as there are with functions), this line between inputs and outputs is intentionally blurred in CLP.

A.2.7 Structures

In addition to integers and atoms, we can also form composite data structures. Composite data structures (or just structures) allow us to hold multiple values at once, much like a tuple. The only real difference from a tuple is that they also have a given name associated with them. To illustrate this, the code snippet below shows a structure named `foo` which contains the values `1` and `2`:

```
foo(1, 2)
```

Structures can be used to build up larger data structures, and even recursive data structures like lists and trees. To demonstrate, we will implement some list operations below which operate on a custom list definition, where `cons` is the name of a structure representing a non-empty list, and `nil` is an atom representing an empty list. With this in mind, we will define a `myAppend` routine which appends two lists together, resulting in a third list. The name "`myAppend`" is used to avoid a conflict with a built-in operation named "`append`", which performs the same basic operation. Without further ado, `myAppend` is defined below:

```
1 myAppend(nil, List, List).
2 myAppend(cons(Head, Tail), List, cons(Head, Rest)) :-
3   myAppend(Tail, List, Rest).
```

A line-by-line explanation of the above code follows:

1. If the first list is empty, the result (held in the third parameter) should be the same as the second list. In other words, if we append an empty list onto some other list `List`, then the result should be `List` (i.e., the list does not change).

2. If the first list is non-empty, name the components of the list `Head` (for the first element of the list) and `Tail` (for the rest of the list). The result list should start with this same `Head`, followed by the other list `Rest`, which has not yet been defined.

3. Recursively call `myAppend`, using `Tail` (the rest of the elements in the first parameter), `List` (the second parameter), and `Rest` (the recursive result).

An example query using the above `myAppend` definition follows. This query appends the list "`1, 2, 3`" onto the list "`4, 5, 6`", yielding the result list "`1, 2, 3, 4, 5, 6`". The query follows:

```
1  ?- myAppend(cons(1, cons(2, cons(3, nil))),
2              cons(4, cons(5, cons(6, nil))),
3              Result).
4  Result = cons(1,
5              cons(2,
6                cons(3,
7                  cons(4,
8                    cons(5,
9                      cons(6,
10                       nil))))))).
```

Lists are very commonly used in CLP. As such, there is built-in notation for lists, which helps improve readability and gets rid of all the extra parentheses above. This notation is translated down into normal structures which behave very similarly to `cons` and `nil` above; in fact, the only differences are in the structure names used. Some notes on the built-in list notation follow:

- `[]`: The empty list
- `[1]`: A list containing one element, namely `1`
- `[1, 2]`: A list containing two elements, namely `1` and `2`. This same pattern can be used for a list of length 3 (e.g., `[1, 2, 3]`), and so on.
- `[Head|Tail]`: A non-empty list that starts with the element `Head`, where the rest of the list is held in `Tail`.
- `[First, Second|Rest]`: A list with a minimum length of 2, where the first element is `First`, the second element is `Second`, and the rest of the elements (as in, all elements after `Second`) are in the list `Rest`.

We can rewrite `myAppend` with this updated list notation. This rewrite (yielding `myAppend2`) is shown below, along with the updated query:

```
1 myAppend2([], List, List).
2 myAppend2([Head|Tail], List, [Head|Rest]) :-
3   myAppend2(Tail, List, Rest).
4
5 ?- myAppend2([1,2,3], [4,5,6], Result).
6 Result = [1, 2, 3, 4, 5, 6].
```

While the code above looks very different from the code before, this behaves in exactly the same way.

Executing "Backwards"

Before concluding this section, there is an important point to make about how `myAppend2` works. In the previous example, we simply appended the lists `[1, 2, 3]` and `[4, 5, 6]` together, yielding the unsurprising result `[1, 2, 3, 4, 5, 6]`. This is entirely expected, and does not show anything particularly interesting about CLP.

What is interesting about `myAppend2` is that we can execute it effectively "in reverse". For example, instead of giving it the known inputs of `[1, 2, 3]` and `[4, 5, 6]` and asking for the output, we can instead give a known output of `[1, 2, 3, 4, 5, 6]` and ask for inputs. Such a query is shown below:

```
?- myAppend2(Input1, Input2, [1, 2, 3, 4, 5, 6]).
```

If we keep hitting semicolon and ask for different query results, we end up with 7 in all, listed below:

1. `Input1 = [], Input2 = [1, 2, 3, 4, 5, 6]`
2. `Input1 = [1], Input2 = [2, 3, 4, 5, 6]`
3. `Input1 = [1, 2], Input2 = [3, 4, 5, 6]`
4. `Input1 = [1, 2, 3], Input2 = [4, 5, 6]`

5. `Input1 = [1, 2, 3, 4], Input2 = [5, 6]`

6. `Input1 = [1, 2, 3, 4, 5], Input2 = [6]`

7. `Input1 = [1, 2, 3, 4, 5, 6], Input2 = []`

As shown with these results, such a query effectively asks "which two unknown lists, when appended to each other, yield the list `[1, 2, 3, 4, 5, 6]`?" Each of the above answers reflects some possible combination of the input lists `Input1` and `Input2` to yield this expected result list (e.g., the first result appends the empty list onto `[1, 2, 3, 4, 5, 6]`, the second result appends the list `[1]` onto the list `[2, 3, 4, 5, 6]`, and so on). In this way, depending on the query we issue to `myAppend2`, we can effectively execute "backwards", going from known outputs to unknown inputs.

A.2.8 Unification

While Section A.2.3 discussed variable usage at length, and variables have been used in each subsequent section, a full discussion of how variables get values has never been provided. These details are important for understanding how CLP works overall, particularly in the context of "backwards execution" in Section A.2.7. We shed some insight on exactly how variables get values in this section.

In CLP, variables are assigned values via *unification*. Unification is effectively mathematical equality (=), but in a way that can be implemented within the CLP engine. In fact, `=` is used to tell the CLP engine to unify two values, just as we would say that two values should be equal in math. Like mathematical equality, unification operates in either direction. For example, consider the following two queries:

```
1 ?- X = 1.
2 X = 1.
3 ?- 1 = X.
4 X = 1.
```

Both of these queries result in the assignment of `1` to `X`, and they fundamentally work in the exact same way. This is decidedly **unlike** the assignment operator from other languages, wherein the variable must be on the left and the value must be on the right. This is somewhat obnoxious because other languages still use the syntactic `=` for assignment, even though assignment bears little connection to its mathematical meaning. In contrast, CLP's unification operator works just like mathematical =, and idiomatic CLP lacks assignment. (As an aside, true assignment is also present in most CLP engines, though its usage is strongly discouraged; no code presented in this dissertation uses assignment.)

Also like mathematical = and unlike assignment, once a variable has a value, the value of the variable can never change. For example, the following query fails:

```
1 ?- X = 1, X = 2.
2 false.
```

At the point of `X = 1` in the above query, CLP stores that the value of variable `X` is `1`. When `X = 2` is encountered, this ultimately triggers failure (leading the engine to report `false` above), as `1` and `2` are not equal to each other.

The related query below does, however, succeed:

```
1 ?- X = 1, X = 1.
2 X = 1.
```

At the first use of `X = 1`, the value of variable `X` is stored to be `1`. At the second use of `X = 1`, we simply check to see if the stored value of `X` is `1`. In this case, the stored value of `X` is equal to `1`, and so the query succeeds with the information that the value of `X` is `1`.

This process is easy enough to follow when we deal with exactly one variable and one value. However, part of the power of CLP is that we can reason about many variables and values. For example, consider the following query:

```
1 ?- X = Y, X = 1, Y = 2.
2 false.
```

The above query fails because we first added a constraint that `X` and `Y` should be the same value, and then we tried to give `X` and `Y` separate values (namely `1` and `2`, respectively). If we had never issued the `X = Y` part in the query above, this query would have succeeded. Such a modified query can be seen below, which succeeds:

```
1 ?- X = 1, Y = 2.
2 X = 1, Y = 2.
```

What the above two queries show is that somehow we need to record if two variables hold the same value, **even if** we don't know exactly what that value is yet. This was the case for the `X = Y` portion above, which recorded that variables `X` and `Y` should hold the same value, even though values had not been given to `X` and `Y`.

This sort of recording can be most easily understood in terms of *equivalence classes*. Roughly, an equivalence class is a set of elements which are all declared to be equal. Initially, all values and variables are in their own separate equivalence classes. For example, let's say that we have variables `X` and `Y`, along with the value `1`. Initially, there are three equivalence classes: one containing `X`, a second containing `Y`, and a third containing `1`. If we issue a unification `X = Y`, this effectively states to *merge* the equivalence classes for `X` and `Y`. This merge leaves us with two equivalence classes: one containing both `X` and `Y` (hereinafter named $X \cup Y$), and a second containing the value `1`. If we then issue the unification `X = 1`, we will then take the equivalence class containing `X` (namely $X \cup Y$) and merge it with the equivalence class which just contains `1`. In this example, the end result is a single equivalence class holding variables `X` and `Y`, along with the value `1`. The semantic meaning of this is that variables `X` and `Y` share the same value, and that value is `1`. For the rest of the discussion, we will call this equivalence class $X \cup Y \cup 1$.

Care must be taken when merging equivalence classes. For example, if we take the

equivalence class $X \cup Y \cup 1$ and attempt to merge it with an equivalence class containing 2, this should fail: 1 does not equal 2, and so we can never merge equivalence classes containing different values. In CLP, such faulty attempts to merge (i.e., attempts to unify values which cannot be unified) lead to failure. This failure will either cause the entire query to fail, or go back to whatever the last choice was and potentially try to unify with a different value.

Appendix B

Formal Notation Used

Throughout this dissertation, mathematical formalisms are employed for a variety of reasons. This appendix explains exactly what this notation means.

B.1 Tuples

Tuples are ordered sequences of elements of possibly different types. Duplicate elements are permitted. The number of elements tuples contain is fixed, though tuples of arbitrarily large size can be constructed.

- A 2-tuple (also known as a pair) of a and b: $(a \cdot b)$
- A 3-tuple (also known as a triple) of a, b, and c: $(a \cdot b \cdot c)$
- A 4-tuple of a, b, c, and d: $(a \cdot b \cdot c \cdot d)$

B.2 Lists

Lists are ordered sequences of elements of a single type. Duplicates are permitted. The number of elements they contain is unbounded.

- Empty list: []
- Sequence of something represented with metavariable a: $\vec{a}$
- Prepending element a_1 on a list $\vec{a_2}$: $a_1 :: \vec{a_2}$
- Getting the number of items in a list $\vec{a}$: $|\vec{a}|$

B.3 Sets

Sets are unordered sequences of elements of a single type. Duplicates are **not** permitted. The number of elements sets contain is unbounded.

- Empty set: {}
- Set of something represented with metavariable a: $\bar{a}$
- Checking if element a is contained in set $\bar{s}$: $a \in \bar{s}$
- Checking if element a is **not** contained in set $\bar{s}$: $a \notin \bar{s}$
- Adding an element a to a set $\bar{s}$: $\bar{s} + a$. If $a \in \bar{s}$, then $\bar{s} + a$ simply returns $\bar{s}$ (effectively a no-op).
- Getting the number of items in a set $\bar{s}$: $|\bar{s}|$. Note that this is the same notation for getting the number of items in a list; whichever this refers to is context-sensitive.

B.4 Maps

Maps are unordered sequences of key/value pairs. Duplicate keys are **not** permitted, though duplicate values (which map to different keys) **are** permitted. The number of key/value pairs maps contain is unbounded.

- Empty map: {}. Note that this is the same notation for empty sets; whichever this refers to is context-senstive.
- The type of a map of key K and value V: $K \to V$. This intentionally is the same notation as a function from K to V, as the two work in effectively the same manner.
- Map lookup, for map m and key k: $m(k)$. This returns the value contained in m for the key k.
- Adding key/value pair consisting of k and v, respectively, to a map m: $m[k \mapsto v]$
- Returning the keys held in a map m as a set: $\texttt{keys}(m)$
- Getting the number of key/value pairs in a map m: $|m|$ Note that this is the same notation for getting the number of items in a list or a set; whichever this refers to is context-sensitive.

B.5 Axioms, Inference Rules, and Typing Rules

When formally describing how a logical system operates, axioms and inference rules will be used. For example, consider the notation used in Figure B.1. The axiom (identified by the name (AXIOM)) states that a is always true. The inference rule (identified by the name (RULE)) states that c is true as long as a and b are true.

$$\overline{a}\ \text{(AXIOM)} \qquad \frac{a \quad b}{c}\ \text{(RULE)}$$

Figure B.1: Example axiom and inference rule notation.

Rules describing type systems build on the above notation. The biggest change is the augmentation of additional symbols that are passed around to keep track of auxilliary information. For example, in a typing context we commonly need to keep track of which

variables are in scope, along with which types those variables are associated with. To better see how this works, an example with a variant of the simply-typed lambda calculus is used. The syntax we will use is shown in Figure B.2. As shown, this grammar adds in arithmetic and relational operations. The corresponding typing rules for this grammar are shown in Figure B.3.

$$n \in \mathbb{N} \quad b \in \mathit{Boolean} \quad x \in \mathit{Variable}$$

$$\begin{aligned} \tau \in \mathit{Type} &::= \mathsf{nat} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \\ e \in \mathit{Exp} &::= x \mid n \mid b \mid \lambda x : \tau \,.\, e \mid e_1(e_2) \\ &\mid e_1 + e_2 \mid e_1 \leq e_2 \end{aligned}$$

Figure B.2: Syntax for a variant of the simply-typed lambda calculus, where $\tau_1 \to \tau_2$ denotes a function type where the input is of type τ_1 and the output is of type τ_2, $\lambda x : \tau \,.\, e$ denotes a function that takes a parameter x of type τ with body e, $e_1(e_2)$ is intended to call a function delineated by e_1 with the parameter delineated by e_2, $e_1 + e_2$ adds the two natural expressions in e_1 and e_2, and $e_1 \leq e_2$ performs an arithmetic less-than-or-equal operation over e_1 and e_2.

$$\Gamma \in \mathit{Variable} \to \mathit{Type}$$

$$\frac{x \in \mathtt{keys}(\Gamma) \quad \tau = \Gamma(x)}{\Gamma \vdash x : \tau}\ (\textsc{var}) \qquad \frac{}{\Gamma \vdash n : \mathsf{nat}}\ (\textsc{nat}) \qquad \frac{}{\Gamma \vdash b : \mathsf{bool}}\ (\textsc{bool})$$

$$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau \,.\, e) : \tau \to \tau'}\ (\textsc{fun}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}\ (\textsc{app})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{nat} \quad \Gamma \vdash e_2 : \mathsf{nat}}{\Gamma \vdash e_1 + e_2 : \mathsf{nat}}\ (\textsc{add}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{nat} \quad \Gamma \vdash e_2 : \mathsf{nat}}{\Gamma \vdash e_1 \leq e_2 : \mathsf{bool}}\ (\textsc{leq})$$

Figure B.3: Typing rules for the syntax shown in Figure B.2.

Figure B.3 adds in *judgements* of the form $\Gamma \vdash e : \tau$, where:

- Γ is a mapping of variables in scope to their corresponding types

- e is an expression
- τ is the type of the expression e, as derived through the typing rules in Figure B.3

The typing rules are used in a recursive manner, reflecting the fact that expressions were recursively defined in Figure B.2. An English description of each of the typing rules in Figure B.3 is provided below:

- (VAR): Gets the type of the variable x
- (NAT): All natural numbers are of type **nat**
- (BOOL): All booleans are of type **bool**
- (FUN): The input type of a function is determined by the annotated type of its parameter, namely τ. The return type of a function is determined by typechecking its body e, resulting in the return type τ'. In order to typecheck e, we must first put the variable x in scope and associate it with its expected type τ.
- (APP): e_1 should be a function type. The expected input type of the function should be the same as the parameter type, where the parameter is specified by e_2. Function calls overall have the same type as whatever the return type of the function is.
- (ADD): Adding two natural numbers yields a natural number
- (LEQ): Comparing two natural numbers to each other yields a boolean

Appendix C

Full System F Generator

The generator code below is based on the code in Chapter 4, Figure 4.3. Unlike the code in Chapter 4, Figure 4.3, the code below immediately works as a generator of well-typed expressions in System F, and it includes the definitions of all helper procedures. Overall, there are several key differences between these two codebases:

- Depth-based bounding is employed (described in Chapter 7, Section 7.3.1). The `initialBound` procedure is used to determine the initial bound to use.
- The values of integers are bound with the help of the `validInteger` procedure. This problem is discussed more thoroughly in Chapter 7, Section 7.3.2.
- Variable and type variable names are bound, with the help of the `validVariable` and `validTypeVariable` procedures, respectively. This problem is disucussed more thoroughly in Chapter 7, Section 7.3.3.
- Cyclic terms are avoided with the help of the built-in `unify_with_occurs_check` procedure. This problem is disucussed more thoroughly in Chapter 7, Section 7.3.4.
- There is an additional "hole-filling" phase which occurs after a well-typed program is

generated, in order to ensure that output programs are fully instantiated. Without this phase, we may emit programs like the following, where `???` represents an uninstantiated logical variable:

$$\lambda x : \texttt{???} \,.\, 42$$

As shown above, the actual type of x is irrelevant to the program being well-typed, as the program does not use x. Because the type is irrelevant, the `typeof` procedure will fail to instantiate it, because at no point during generation will the type become constrained. The subsequent hole-filling pass will ensure that any unconstrained types are filled in with an arbitrary type, specifically **integer** in the implementation below. This is similar to the same problem as filling in "holes" for integers, discussed in Chapter 7, Section 7.3.2.

Wihout further ado, the code is presented below in its entirety. The last lines (137-139) give a query which will generate all well-typed programs up to depth `2` (specified by the `initialBound` procedure) in our variant of System F.

```
 1 initialBound(2).
 2
 3 validInteger(0).
 4
 5 validVariable(x).
 6 validVariable(y).
 7 validVariable(z).
 8
 9 validTypeVariable('A').
10 validTypeVariable('B').
11 validTypeVariable('C').
12
13 decBound(Bound, NewBound) :-
14   Bound > 0,
15   NewBound is Bound - 1.
16
17 lookupMap([pair(Var, Type1)|_], Var, Type2) :-
18   validVariable(Var),
```

```
19    unify_with_occurs_check(Type1, Type2).
20  lookupMap([OtherVar|Rest], Var, Type) :-
21    validVariable(OtherVar),
22    validVariable(Var),
23    OtherVar \== Var,
24    lookup(Rest, Var, Type).
25
26  addMap(X, T, [], [pair(X, T)]) :-
27    validVariable(X).
28  addMap(X, T, [pair(X, _)|Rest], [pair(X, T)|Rest]) :-
29    validVariable(X).
30  addMap(X1, T,
31          [NonMatchPair|OldRest],
32          [NonMatchPair|NewRest]) :-
33    NonMatchPair = pair(X2, _),
34    validVariable(X1),
35    validVariable(X2),
36    X1 \== X2,
37    addMap(X1, T, OldRest, NewRest).
38
39  % substitute occurrances of A with T1 in T2 to yield T3
40  substitute(A, T1, T2, T3) :-
41    validTypeVariable(A),
42    initialBound(Bound),
43    substitute(Bound, A, T1, T2, T3).
44
45  % substitute: Bound, A, T1, T2, T3
46  substitute(_, _, _, integer, integer).
47  substitute(_, TypeVar,
48             ReplaceWith, TypeVar, ReplaceWith) :-
49    validTypeVariable(TypeVar).
50  substitute(Bound, TypeVar, ReplaceWith,
51             arrow(T1, T2),
52             arrow(NewT1, NewT2)) :-
53    validTypeVariable(TypeVar),
54    decBound(Bound, NewBound),
55    substitute(NewBound, TypeVar, ReplaceWith, T1, NewT1),
56    substitute(NewBound, TypeVar, ReplaceWith, T2, NewT2).
57  substitute(Bound, TypeVar, ReplaceWith,
58             poly(OtherTypeVar, T),
59             poly(OtherTypeVar, NewT)) :-
60    validTypeVariable(TypeVar),
```

```
61    validTypeVariable(OtherTypeVar),
62    decBound(Bound, NewBound),
63    (TypeVar == OtherTypeVar ->
64      T = NewT; % shadowing occurs
65      substitute(NewBound, TypeVar, ReplaceWith, T, NewT)).
66
67  typeof(Gamma, E, T) :-
68    initialBound(Bound),
69    typeof(Bound, Gamma, E, T).
70
71  typeof(_, _, int(I), integer) :-
72    validInteger(I).
73  typeof(_, Gamma, variable(X), T) :-
74    validVariable(X),
75    lookupMap(Gamma, X, T).
76  typeof(Bound, Gamma, lam(X, T1, E), arrow(T1, T2)) :-
77    decBound(Bound, NewBound),
78    validVariable(X),
79    addMap(X, T1, Gamma, NewGamma),
80    typeof(NewBound, NewGamma, E, T2).
81  typeof(Bound, Gamma, app(E1, E2), T2) :-
82    decBound(Bound, NewBound),
83    typeof(NewBound, Gamma, E1, arrow(T1, T2)),
84    typeof(NewBound, Gamma, E2, T1).
85  typeof(Bound, Gamma, tlam(A, E), poly(A, T)) :-
86    decBound(Bound, NewBound),
87    validTypeVariable(A),
88    typeof(NewBound, Gamma, E, T).
89  typeof(Bound, Gamma, tapp(E, T1), T3) :-
90    decBound(Bound, NewBound),
91    typeof(NewBound, Gamma, E, poly(A, T2)),
92    validTypeVariable(A),
93    substitute(A, T1, T2, T3).
94
95  % fillHolesHelper: Exp, SubExps, Types
96  fillHolesHelper(variable(X), [], []) :-
97    validVariable(X).
98  fillHolesHelper(lam(X, T, E), [E], [T]) :-
99    validVariable(X).
100 fillHolesHelper(app(E1, E2), [E1, E2], []).
101 fillHolesHelper(tlam(A, E), [E], []) :-
102   validTypeVariable(A).
```

```
103 fillHolesHelper(tapp(E, T), [E], [T]).
104
105 fillTypeHelper(arrow(T1, T2), [T1, T2]).
106 fillTypeHelper(poly(A, T), [T]) :-
107   validTypeVariable(A).
108
109 fillTypeHoles(Type) :-
110   ground(Type).
111 fillTypeHoles(Type) :-
112   var(Type),
113   Type = integer.
114 fillTypeHoles(Type) :-
115   nonvar(Type),
116   \+ ground(Type),
117   fillTypeHelper(Type, SubTypes),
118   maplist(fillTypeHoles, SubTypes).
119
120 fillExpHoles(Exp) :-
121   ground(Exp).
122 fillExpHoles(Exp) :-
123   var(Exp),
124   validInteger(I),
125   Exp = int(I).
126 fillExpHoles(Exp) :-
127   nonvar(Exp),
128   \+ ground(Exp),
129   fillHolesHelper(Exp, SubExps, Types),
130   maplist(fillTypeHoles, Types),
131   maplist(fillExpHoles, SubExps).
132
133 wellTyped(E, T) :-
134   typeof([], E, T),
135   fillExpHoles(E).
136
137 ?- wellTyped(E, T),
138    writeln(E),
139    fail.
```

Appendix D

Helper Functions in Typed-Prolog

`envVariableType`

If the given variable is contained in the given type environment, then it returns the type of that variable. Otherwise, it adds the variable to the type environment, and yields some arbitrary type. While this is written as nondeterministically returning all possible types (via τ), in the implementation unification (as already exists in CLP) is used to handle this efficiently.

$\texttt{envVariableType} \in \mathit{TypeEnv} \times \mathit{Variable} \to (\mathit{Type} \times \mathit{TypeEnv})$

$$\texttt{envVariableType}(\Gamma, x) = \begin{cases} (\Gamma(x), \Gamma) & \text{if } x \in \texttt{keys}(\Gamma) \\ (\tau, \Gamma[x \mapsto \tau]) & \text{otherwise} \end{cases}$$

typeofTerms

Given a type environment and a list of terms, returns a list of types of the same size, where each type corresponds to each input term. Also returns an output type environment, resulting from chaining through the type environment while determining the type of each term.

$\texttt{typeofTerms} \in \mathit{TypeEnv} \times \overrightarrow{\mathit{Term}} \to (\overrightarrow{\mathit{Type}} \times \mathit{TypeEnv})$

$\texttt{typeofTerms}(\Gamma, []) = ([], \Gamma)$

$\texttt{typeofTerms}(\Gamma_1, \mathit{term}_1 :: \overrightarrow{\mathit{term}_2}) =$

$\Gamma_1 \vdash \mathit{term}_1 : \tau_1 \cdot \Gamma_2$

let $(\Gamma_3, \vec{\tau_2}) = \texttt{typeofTerms}(\Gamma_2, \overrightarrow{\mathit{term}_2})$

$(\tau_2 :: \vec{\tau_2}, \Gamma_3)$

unifyTypes

Takes the following:

- A template type containing type variables (expected type)
- The type to unify with (received type)
- A mapping of type variables to types they have already been unified with

Returns a new type variable mapping. The metavariable m is used to represent the mapping of $\mathit{TypeVariable} \to \mathit{Type}$. Gets stuck if a type error occurs.

$\texttt{unifyTypes} \in \mathit{Type} \times \mathit{Type} \times (\mathit{TypeVariable} \to \mathit{Type}) \to (\mathit{TypeVariable} \to \mathit{Type})$

$\texttt{unifyTypes}(\textsf{int}, \textsf{int}, m) = m$

$\texttt{unifyTypes}(\textsf{atom}, \textsf{atom}, m) = m$

$\texttt{unifyTypes}(\textsf{bool}, \textsf{bool}, m) = m$

$\texttt{unifyTypes}(\textsf{relation}(\vec{\tau_1}), \textsf{relation}(\vec{\tau_2}), m) =$

$\quad \texttt{unifyTypesList}(\vec{\tau_1}, \vec{\tau_2}, m)$

$\texttt{unifyTypes}(un(\vec{\tau_1}), un(\vec{\tau_2}), m) =$

$\quad \texttt{unifyTypesList}(\vec{\tau_1}, \vec{\tau_2}, m)$

$\texttt{unifyTypes}(T, \tau) =$

$$\begin{cases} m[T \mapsto \tau] & \text{if } T \notin \texttt{keys}(m) \\ m & \text{if } T \in \texttt{keys}(m) \wedge m(T) = \tau \end{cases}$$

unifyTypesList

Lifts `unifyTypes` to lists of types. The metavariable m is used to represent the mapping of $\mathit{TypeVariable} \to \mathit{Type}$.

$\texttt{unifyTypesList} \in \overrightarrow{\mathit{Type}} \times \overrightarrow{\mathit{Type}} \times (\mathit{TypeVariable} \to \mathit{Type}) \to (\mathit{TypeVariable} \to \mathit{Type})$

$\texttt{unifyTypesList}([], [], m) = m$

$\texttt{unifyTypesList}(\tau_1 :: \vec{\tau_2}, \tau_3 :: \vec{\tau_4}, m_1) =$

$\quad$ let $m_2 = \texttt{unifyTypes}(\tau_1, \tau_3, m_1)$

$\quad$ $\texttt{unifyTypesList}(\vec{\tau_2}, \vec{\tau_4}, m_2)$

unifyPolyHelper

The metavariable m is used to represent the mapping of $\mathit{TypeVariable} \to \mathit{Type}$. Mathematically, τ in the helper below denotes nondeterministic choice between all possible types. In the implementation this is handled efficiently thanks to unification; a fresh, uninstantiated logical variable is used in place of τ.

$\texttt{unifyPolyHelper} \in \overrightarrow{\mathit{TypeVariable}} \times (\mathit{TypeVariable} \to \mathit{Type}) \to \overrightarrow{\mathit{Type}}$

$\texttt{unifyPolyHelper}([], m) = []$

$\texttt{unifyPolyHelper}(T_1 :: \vec{T_2}, m) =$

$\quad$ let $\tau = \begin{cases} m(T) & \text{if } T \in \texttt{keys}(m) \\ \tau & \text{otherwise} \end{cases}$

$\quad$ $\tau :: \texttt{unifyPolyHelper}(\vec{T_2}, m)$

unifyPoly

Takes the following:

- The list of type variables in play
- A list of expected types, which may contain the type variables held in the first parameter
- A list of actual types, which should be unified with the expected types

Returns a list of types which correspond one-to-one with the input list of type variables. For example, consider the following call to `unifyPoly` and corresponding return value:

$\texttt{unifyPoly}([T], [\textsf{relation}(T, T)], [\textsf{relation}(\textsf{int}, \textsf{int})]) = \{T \mapsto \textsf{int}\}$

The metavariable m is used to represent the mapping of $\mathit{TypeVariable} \rightarrow \mathit{Type}$. In the actual implementation, this operation is replaced with unification between the two lists of types, treating the type variables as logic variables. This ends up implicitly building the output type mapping in the process.

$\texttt{unifyPoly} \in \overrightarrow{\mathit{TypeVariable}} \times \overrightarrow{\mathit{Type}} \times \overrightarrow{\mathit{Type}} \rightarrow \overrightarrow{\mathit{Type}}$

$\texttt{unifyPoly}(\overrightarrow{T}, \vec{\tau_1}, \vec{\tau_2}) =$

$\quad$ let $m = \texttt{unifyTypesList}(\vec{\tau_1}, \vec{\tau_2}, \{\})$

$\quad \texttt{unifyPolyHelper}(\overrightarrow{T}, m)$

Appendix E

Helper Functions in SimpleScala

`makeMap`

Given two sequences of the same length, creates a map out of them where the first sequence contains keys, and the second sequence contains values. Gets stuck if the two sequences are of different lengths. If the keys contain duplicate elements, then the first key in the first sequence will "win out" and end up in the resulting map with whatever value it maps to.

$\texttt{makeMap} \in \vec{K} \times \vec{V} \to (K \to V)$

$\texttt{makeMap}([], []) = \{\}$

$\texttt{makeMap}(k_1 :: \vec{k_2}, v_1 :: \vec{v_2}) =$

$\quad (\texttt{makeMap}(\vec{k_2}, \vec{v_2}))[k_1 \mapsto v_1]$

typeOk

Returns `true` if all the type variables in the given type are in scope.

$\texttt{typeOk} \in \mathit{Type} \to \mathit{Boolean}$

$\texttt{typeOk}(\textbf{string}) = \textbf{true}$

$\texttt{typeOk}(\textbf{boolean}) = \textbf{true}$

$\texttt{typeOk}(\textbf{integer}) = \textbf{true}$

$\texttt{typeOk}(\textbf{unitType}) = \textbf{true}$

$\texttt{typeOk}(\tau_1 \to \tau_2) =$

$\quad \texttt{typeOk}(\tau_1) \wedge \texttt{typeOk}(\tau_2)$

$\texttt{typeOk}((\ \vec{\tau}\ )) =$

$\quad \texttt{typeOkList}(\vec{\tau})$

$\texttt{typeOk}(\mathit{un}[\vec{\tau}]) =$

$\quad \texttt{typeOkList}(\vec{\tau})$

$\texttt{typeOk}(T) = T \in \mathit{tscope}$

typeOkList

Returns **true** if `typeOk` returns **true** on each input type.

$\texttt{typeOkList} \in \overrightarrow{\mathit{Type}} \to \mathit{Boolean}$

$\texttt{typeOkList}([]) = \textbf{true}$

$\texttt{typeOkList}(\tau_1 :: \vec{\tau_2}) =$

$\quad \texttt{typeOk}(\tau_1) \wedge \texttt{typeOkList}(\vec{\tau_2})$

typeReplace

Performs type replacement. The first parameter is the type variables in the play. The second parameter is the types these type variables should map to. The third parameter is the type on which to perform type replacement. Returns the resulting replaced type. To illustrate, consider the following examples, where A and B are type variables:

$\texttt{typeReplace}([A], [\textsf{integer}], \textsf{string}) = \textsf{string}$

$\texttt{typeReplace}([A], [\textsf{integer}], A) = \textsf{integer}$

$\texttt{typeReplace}([A], [\textsf{integer}], A \rightarrow (\ A, \textsf{string}\ )) = \textsf{integer} \rightarrow (\ \textsf{integer}, \textsf{string}\ )$

$\texttt{typeReplace}([A], [\textsf{integer}], A \rightarrow B) = \textsf{integer} \rightarrow B$

Gets stuck if the number of type variables in the first parameter differs from the number of types provided in the second parameter.

$\texttt{typeReplace} \in \overrightarrow{\mathit{TypeVariable}} \times \overrightarrow{\mathit{Type}} \times \mathit{Type} \rightarrow \mathit{Type}$

$\texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2) =$

$\quad \texttt{typeReplaceHelper}(\texttt{makeMap}(\overrightarrow{T}, \vec{\tau_1}), \tau_2)$

typeReplaceHelper

Helper function used by `typeReplace`. Actually performs the traversal over types.

$\texttt{typeReplaceHeader} \in (\mathit{TypeVariable} \to \mathit{Type}) \times \mathit{Type} \to \mathit{Type}$

$\texttt{typeReplaceHelper}(m, \textsf{\textbf{string}}) = \textsf{\textbf{string}}$

$\texttt{typeReplaceHelper}(m, \textsf{\textbf{boolean}}) = \textsf{\textbf{boolean}}$

$\texttt{typeReplaceHelper}(m, \textsf{\textbf{integer}}) = \textsf{\textbf{integer}}$

$\texttt{typeReplaceHelper}(m, \textsf{\textbf{unitType}}) = \textsf{\textbf{unitType}}$

$\texttt{typeReplaceHelper}(m, \tau_1 \to \tau_2) =$

$\quad \text{let } \tau_1' = \texttt{typeReplaceHelper}(m, \tau_1)$

$\quad \text{let } \tau_2' = \texttt{typeReplaceHelper}(m, \tau_2)$

$\quad \tau_1' \to \tau_2'$

$\texttt{typeReplaceHelper}(m, (\ \vec{\tau_1}\ )) =$

$\quad \text{let } \vec{\tau_2} = \texttt{typeReplaceHelperList}(m, \vec{\tau_1})$

$\quad (\ \vec{\tau_2}\ )$

$\texttt{typeReplaceHelper}(m, un[\vec{\tau_1}]) =$

$\quad \text{let } \vec{\tau_2} = \texttt{typeReplaceHelperList}(m, \vec{\tau_1})$

$\quad un[\vec{\tau_2}]$

$\texttt{typeReplaceHelper}(m, T) = m(T)$

typeReplaceHelperList

Helper function used by `typeReplaceHelper`, specifically for handling lists of types. This effectively just performs a functional `map` operation, without the higher-order function.

$\texttt{typeReplaceHelperList} \in (\mathit{TypeVariable} \to \mathit{Type}) \times \overrightarrow{\mathit{Type}} \to \overrightarrow{\mathit{Type}}$

$\texttt{typeReplaceHelperList}(m, []) = []$

$\texttt{typeReplaceHelperList}(m, \tau_1 :: \vec{\tau_2}) =$

$\quad$ let $\tau_1' = \texttt{typeReplaceHelper}(m, \tau_1)$

$\quad$ let $\vec{\tau_2}' = \texttt{typeReplaceHelperList}(m, \vec{\tau_2})$

$\quad \tau_1' :: \vec{\tau_2}'$

blockEnv

Given a list of *Val* definitions along with an type environment, returns a new type environment where all the provided *Val* definitions are in scope. The name of this helper reflects the fact that this is used for typechecking blocks in SimpleScala, which consist of a series of *Val* definitions.

$\texttt{blockEnv} \in \overrightarrow{\mathit{Val}} \times \mathit{TypeEnv} \to \mathit{TypeEnv}$

$\texttt{blockEnv}([], \Gamma) = \Gamma$

$\texttt{blockEnv}((\textsf{val}\ x = e) :: \overrightarrow{val}, \Gamma) =$

$\quad \Gamma \vdash e : \tau$

$\quad \texttt{blockEnv}(\overrightarrow{val}, \Gamma[x \mapsto \tau])$

tupleTypes

Given a list of expressions, gets each one of their types.

$\texttt{tupleTypes} \in \overrightarrow{Exp} \times TypeEnv \to \overrightarrow{Type}$

$\texttt{tupleTypes}([], \Gamma) = []$

$\texttt{tupleTypes}(e_1 :: \vec{e_2}, \Gamma) =$

$\Gamma \vdash e_1 : \tau$

$\tau :: \texttt{tupleTypes}(\vec{e_2}, \Gamma)$

tupleAccess

Accesses the n^{th} element of a list. The list is one-indexed.

$\texttt{tupleAccess} \in \vec{A} \times \mathbb{N} \to A$

$\texttt{tupleAccess}(a_1 :: \vec{a_2}, n) =$

$$\begin{cases} a_1 & \text{if } n = 1 \\ \texttt{tupleAccess}(\vec{a_2}, n-1) & \text{if } n > 1 \end{cases}$$

tupGamma

Takes a list of variables and a list of types, which are required to be of the same length. Puts each variable into scope with the corresponding type, updating the given type environment. This returns a new type environment reflecting the collective updates.

$\texttt{tupGamma} \in \overrightarrow{\mathit{Variable}} \times \overrightarrow{\mathit{Type}} \times \mathit{TypeEnv} \to \mathit{TypeEnv}$

$\texttt{tupGamma}([], [], \Gamma) = \Gamma$

$\texttt{tupGamma}(x_1 :: \vec{x_2}, \tau_1 :: \vec{\tau_2}, \Gamma) =$

$\quad \texttt{tupGamma}(\vec{x_2}, \vec{\tau_2}, \Gamma[x_1 \mapsto \tau_1])$

casesOk

Returns **true** if all of the following are true:

1. Each constructor name used in the cases is mentioned in the user-defined type for the case
2. No two cases share the same name
3. No case mentioned in the user-defined types is missing in the list of cases

$\texttt{casesOk} \in \overrightarrow{\mathit{Case}} \times \mathit{UserDefinedTypeName} \to \mathit{Boolean}$

$\texttt{casesOk}(\overrightarrow{\mathit{case}}, \mathit{un}) =$

$\quad \text{let } (\overrightarrow{T}, m) = \mathit{tdefs}(\mathit{un})$

$\quad \texttt{casesOkHelper}(\overrightarrow{\mathit{case}}, \{\}, m)$

casesOkHelper

Helper function used by `casesOk`.

$$\texttt{casesOkHelper} \in \overrightarrow{Case} \times \overrightarrow{ConstructorName} \times (ConstructorName \to Type) \to Boolean$$

$$\texttt{casesOkHelper}([], \overline{cn}, m) = (\texttt{keys}(m) = \overline{cn})$$

$$\texttt{casesOkHelper}(case_1 :: \overrightarrow{case_2}, \overline{cn_2}, m) =$$

$$(case_1 = (\textsf{case}\ cn_1(x) \Rightarrow e) \wedge$$

$$cn_1 \in \texttt{keys}(m) \wedge$$

$$cn_1 \notin \overline{cn_2} \wedge$$

$$\texttt{casesOkHelper}(\overrightarrow{case}, \overline{cn_2} + cn_1, m))$$

casesTypes

Gets the type of the expression for each case in a given list of cases.

$$\texttt{casesTypes} \in \overrightarrow{Case} \times TypeEnv \times \overrightarrow{TypeVariable} \times \overrightarrow{Type} \times$$
$$(ConstructorName \to Type) \to \overrightarrow{Type}$$

$$\texttt{casesTypes}([], \Gamma, \overrightarrow{T}, \vec{\tau}, m) = []$$

$$\texttt{casesTypes}((\textsf{case}\ cn(x) \Rightarrow e) :: \overrightarrow{case}, \Gamma, \overrightarrow{T}, \vec{\tau_1}, m) =$$

$$\text{let } \tau_2 = m(cn)$$

$$\text{let } \tau_2' = \texttt{typeReplace}(\overrightarrow{T}, \vec{\tau_1}, \tau_2)$$

$$\Gamma[x \mapsto \tau_2'] \vdash e : \tau_3$$

$$\tau_3 :: \texttt{casesTypes}(\overrightarrow{case}, \Gamma, \overrightarrow{T}, \vec{\tau_1}, m)$$

asSingleton

Given a list that is expected to be of all the same elements, returns any one of the (identical) elements. Gets stuck if the list is not all of the same element, or if the list is empty. For example, `asSingleton`$([1,1,1]) = 1$, but `asSingleton`$([1,1,2,2])$ gets stuck.

`asSingleton` $\in \vec{A} \to A$

`asSingleton`$(a :: []) = a$

`asSingleton`$(a_1 :: a_1 :: \vec{a_2}) =$

    `asSingleton`$(a_1 :: \vec{a_2})$

Bibliography

[1] M. Dowson, "The ariane 5 software failure," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 2, pp. 84–, Mar. 1997. [Online]. Available: http://doi.acm.org/10.1145/251880.251992

[2] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993. [Online]. Available: http://dx.doi.org/10.1109/MC.1993.274940

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/96267.96279

[4] "Automated penetration testing with white-box fuzzing." [Online]. Available: https://msdn.microsoft.com/en-us/library/cc162782.aspx

[5] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[6] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *Proceedings of the 2nd international conference on Tests and proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792786.1792798

[7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[9] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java," in *NASA Formal Methods*, ser. NASA Conference Proceedings, E. Denney, D. Giannakopoulou, and C. S. Pasareanu, Eds., vol. NASA/CP-2009-215407, 2009, pp. 121–125.

[10] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[11] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[13] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

[14] J. Ruderman, "Introducing jsfunfuzz," 2007. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[15] M. Zalewski, "Announcing cross_fuzz: a potential 0-day in circulation, and more," 2011. [Online]. Available: http://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html

[16] J. Ruderman, "Jesse's fuzzer for arithmetic correctness in tracemonkey," 2008. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=465274

[17] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: ACM, 2009, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1670412.1670413

[18] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 65–76. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737986

[19] "Automated generation of test inputs." [Online]. Available: https://blog.regehr.org/archives/340

[20] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 85–99. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908095

[21] W. M. McKeeman, "Differential testing for software." *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[22] "[meta] bugs found by jsfunfuzz," 2006. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=349611

[23] "Langfuzz (grammar-based mutation fuzzer) - js shell bugs," 2011. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=676763

[24] V. St-Amour and N. Toronto, "Experience report: applying random testing to a base type environment," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 351–356. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500616

[25] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287651

[26] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005, pp. 504–527.

[27] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.602

[28] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," Department of Computer Science, The University of Iowa, Tech. Rep., 2015, available at `www.SMT-LIB.org`.

[29] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 281–294. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065045

[30] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter, "Synthesis modulo recursive functions," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 407–426. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509555

[31] P.-M. Osera and S. Zdancewic, "Type-and-example-directed program synthesis," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 619–630. [Online]. Available: http://doi.acm.org/10.1145/2737924.2738007

[32] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 22–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872551

[33] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149

[34] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 123–133. [Online]. Available: http://doi.acm.org/10.1145/566172.566191

[35] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806835

[36] K. Dewey, L. Nichols, and B. Hardekopf, "Automated data structure generation: Refuting common wisdom," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 32–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818761

[37] N. Rosner, V. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid, "Bounded exhaustive test input generation from hybrid invariants," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 655–674. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660232

[38] V. Senni and F. Fioravanti, "Generation of test data structures using constraint logic programming," in *Proceedings of the 6th international conference on Tests and Proofs*, ser. TAP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 115–131. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30473-6_10

[39] F. Fioravanti, M. Proietti, and V. Senni, "Efficient generation of test data structures using constraint logic programming and program transformation," *Journal of Logic Computation*, 2013.

[40] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 111–119. [Online]. Available: http://doi.acm.org/10.1145/41625.41635

[41] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *Journal of Logic Programming*, vol. 19, pp. 503–581, 1994.

[42] B. Fetscher, K. Claessen, M. Palka, J. Hughes, and R. B. Findler, "Making random judgements: Automatically generating well-typed terms from the definition of a type-system," eSOP 2015.

[43] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, "Run your research: On the effectiveness of lightweight mechanization," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 285–296. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103691

[44] M. Flatt and PLT, "Reference: Racket," Tech. Rep. PLT-TR-2010-1, 2010. [Online]. Available: http://racket-lang.org/tr1/

[45] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. [Online]. Available: http://books.google.com/books?id=anJsH3Dq5BIC

[46] W. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Seldin and J. Hindley, Eds. Academic Press, 1980, pp. 479–490.

[47] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[48] T. Z. Team, Personal Communication.

[49] "Acm digital library." [Online]. Available: http://dl.acm.org/

[50] C. Barrett, L. de Moura, and A. Stump, "Smt-comp: Satisfiability modulo theories competition," in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 20–23. [Online]. Available: http://dx.doi.org/10.1007/11513988_4

[51] C. Barrett, M. Deters, L. Moura, A. Oliveras, and A. Stump, "6 years of smt-comp," *J. Autom. Reason.*, vol. 50, no. 3, pp. 243–277, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10817-012-9246-5

[52] T. Toda and T. Soh, "Implementing efficient all solutions sat solvers," *J. Exp. Algorithmics*, vol. 21, pp. 1.12:1–1.12:44, Nov. 2016. [Online]. Available: http://doi.acm.org/10.1145/2975585

[53] J. Hooker, "Solving the incremental satisfiability problem," *The Journal of Logic Programming*, vol. 15, no. 1âĂŞ2, pp. 177 – 186, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/074310669390018C

[54] J. Whittemore, J. Kim, and K. Sakallah, "Satire: A new incremental satisfiability engine," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 542–545. [Online]. Available: http://doi.acm.org/10.1145/378239.379019

[55] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, Jul. 1960. [Online]. Available: http://doi.acm.org/10.1145/321033.321034

[56] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[57] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/357073.357079

[58] V. W. Marek and M. Truszczyński, *Stable Models and an Alternative Logic Programming Paradigm.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 375–398. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-60085-2_17

[59] Y. Dimopoulos, B. Nebel, and J. Koehler, *Encoding planning problems in nonmonotonic logic programs.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 169–181. [Online]. Available: http://dx.doi.org/10.1007/3-540-63912-8_84

[60] D. H. D. Warren, L. M. Pereira, and F. Pereira, "Prolog - the language and its implementation compared with lisp," *SIGART Bull.*, no. 64, pp. 109–115, Aug. 1977. [Online]. Available: http://doi.acm.org/10.1145/872736.806939

[61] P. Roussel, *Prolog Manual de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle der Marseille Lumimy, 1975. [Online]. Available: http://books.google.com/books?id=Yj7qMgEACAAJ

[62] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński, *The Second Answer Set Programming Competition.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 637–654. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04238-6_75

[63] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, vol. 187-188, pp. 52–89, 2012.

[64] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, *clasp: A Conflict-Driven Answer Set Solver.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 260–265. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72200-7_23

[65] M. Carlsson and P. Mildner, "Sicstus prolog-the first 25 years," *Theory Pract. Log. Program.*, vol. 12, no. 1-2, pp. 35–66, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1017/S1471068411000482

[66] D. Diaz and P. Codognet, "The gnu prolog system and its implementation," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, ser. SAC '00. New York, NY, USA: ACM, 2000, pp. 728–732. [Online]. Available: http://doi.acm.org/10.1145/338407.338553

[67] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[68] M. Wallace, S. Novello, and J. Schimpf, "Eclipse: A platform for constraint logic programming," 1997.

[69] U. Nilsson and J. Maluszynski, "Logic, programming, and prolog (second edition)," http://www.ida.liu.se/~ulfni/lpp/bok/bok.pdf.

[70] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 725–730. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642963

[71] Mozilla, "The rust language website." [Online]. Available: http://www.rust-lang.org/

[72] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 482–493. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.65

[73] K. Dewey, P. Conrad, M. Craig, and E. Morozova, "Evaluating test suite effectiveness and assessing student code via constraint logic programming," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '17. New York, NY, USA: ACM, 2017, pp. 317–322. [Online]. Available: http://doi.acm.org/10.1145/3059009.3059051

[74] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification." in *ICSE*. IEEE Computer Society, 2007, pp. 621–631. [Online]. Available: http://dblp.uni-trier.de/db/conf/icse/icse2007.html#GroceHJ07

[75] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[76] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 112–122. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250748

[77] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: ACM, 2002, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/587051.587053

[78] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[79] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254104

[80] J. Ruderman, "Introducing lithium, a testcase reduction tool," 2007. [Online]. Available: http://www.squarefree.com/2007/09/15/introducing-lithium-a-testcase-reduction-tool/

[81] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *Proceedings of the 16th IEEE International Symposium on Software Reliability*

Engineering, ser. ISSRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–276. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2005.28

[82] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 417–420. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321698

[83] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engg.*, vol. 10, no. 2, pp. 203–232, Apr. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1022920129859

[84] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[85] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software assurance by bounded exhaustive testing," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 133–142. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007531

[86] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An evaluation of exhaustive testing for data structures," MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, Tech. Rep., 2003.

[87] V. Jagannath, Y. Y. Lee, B. Daniel, and D. Marinov, "Reducing the costs of bounded-exhaustive testing," in *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ser. FASE '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00593-0_12

[88] E. J. G. Arias, J. M. Carballo, and J. M. R. Poza, "A proposal for disequality constraints in curry," *Electronic Notes in Theoretical Computer Science*, vol. 177, no. 0, pp. 269 – 285, 2007, proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming (WFLP 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066107002265

[89] J. Houghtaling, "Windows 95 animated cursor format." [Online]. Available: http://www.wotsit.org/download.asp?f=ani&sc=463191359

[90] M. Christakis and P. Godefroid, "Proving memory safety of the ani windows image parser using compositional exhaustive testing," Tech. Rep. MSR-TR-2013-120,

November 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=204915

[91] W. Pugh, "Concurrent maintenance of skip lists," College Park, MD, USA, Tech. Rep., 1990.

[92] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3828.3835

[93] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70. New York, NY, USA: ACM, 1970, pp. 107–141. [Online]. Available: http://doi.acm.org/10.1145/1734663.1734671

[94] R. Bayer, "Binary b-trees for virtual memory," in *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '71. New York, NY, USA: ACM, 1971, pp. 219–235. [Online]. Available: http://doi.acm.org/10.1145/1734714.1734731

[95] "Haskell state monad documentation." [Online]. Available: https://wiki.haskell.org/State_Monad

[96] "The gnu prolog website." [Online]. Available: http://www.gprolog.org/

[97] "Publically available rust packages." [Online]. Available: https://crates.io/

[98] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721503

[99] U. S. Reddy, "A typed foundation for directional logic programming," in *In Proc. Workshop on Extensions to Logic Programming*, 1992, pp. 199–222.

[100] L. Sterling and E. Shapiro, *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. Cambridge, MA, USA: MIT Press, 1994.

[101] K. L. Clark, "Readings in nonmonotonic reasoning," M. L. Ginsberg, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, ch. Negation As Failure, pp. 311–325. [Online]. Available: http://dl.acm.org/citation.cfm?id=42641.42664

[102] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.

[103] D. Pearce and G. Wagner, "Logic programming with strong negation," in *Extensions of Logic Programming*, ser. Lecture Notes in Computer Science, P. Schroeder-Heister, Ed. Springer Berlin Heidelberg, 1991, vol. 475, pp. 311–326. [Online]. Available: http://dx.doi.org/10.1007/BFb0038700

[104] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[105] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 78–88. [Online]. Available: http://doi.acm.org/10.1145/04000800.2336763

[106] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 70–81. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931056

[107] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 60–76. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

[108] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 115–134. [Online]. Available: http://doi.acm.org/10.1145/949305.949317

[109] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow, "Associated types with class," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040306

[110] "Rust rfc for associated types and items." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0195-associated-items.md

[111] J. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, pp. 1–102, 1987. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(87)90045-4

[112] P. Wadler, "Linear types can change the world!" in *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[113] D. Gay and A. Aiken, "Memory management with explicit regions," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 313–323. [Online]. Available: http://doi.acm.org/10.1145/277650.277748

[114] K. Crary, D. Walker, and G. Morrisett, "Typed memory management in a calculus of capabilities," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99. New York, NY, USA: ACM, 1999, pp. 262–275. [Online]. Available: http://doi.acm.org/10.1145/292540.292564

[115] J. A. Tov and R. Pucella, "Practical affine types," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926436

[116] "Rust language constraint solver." [Online]. Available: https://doc.rust-lang.org/1.0.0-alpha/rustc_trans/middle/traits/struct.FulfillmentContext.html

[117] A. P. Felty, "Specifying and implementing theorem provers in a higher-order logic programming language," Tech. Rep., 1989.

[118] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*, 2nd ed. USA: Artima Incorporation, 2011.

[119] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 197–208. [Online]. Available: http://doi.acm.org/10.1145/2462156.2462173

[120] "Rfc: No unused impl parameters." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0447-no-unused-impl-parameters.md

[121] "Implied bounds: Rust does not always infer everything possible." [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2014/07/06/implied-bounds/

[122] "New projections, lifetimes, and well-formedness rfc indicates check was previously not performed." [Online]. Available: https://github.com/nikomatsakis/rfcs/blob/projection-and-lifetimes/text/0000-projections-lifetimes-and-wf.md

[123] "Type bounds are conditionally enforced on impls." [Online]. Available: https://github.com/rust-lang/rust/issues/25110

[124] "Bug in checker for duplicate constraints." [Online]. Available: https://github.com/rust-lang/rust/issues/18693

[125] "Same trait with different input concrete parameters are considered duplicate bounds." [Online]. Available: https://github.com/rust-lang/rust/issues/22279

[126] "Box and in for stdlib: Ambiguity involving box and expressions surrounded by parentheses." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0809-box-and-in-for-stdlib.md

[127] "Coherence failed to report ambiguity ice involving lifetimes." [Online]. Available: https://github.com/rust-lang/rust/issues/24424

[128] "Ice using associated type in trait bound." [Online]. Available: https://github.com/rust-lang/rust/issues/20220

[129] "Rfc: Variance: The need for phantomdata." [Online]. Available: https://github.com/nikomatsakis/rfcs/blob/variance-redux/text/0000-variance.md

[130] "Type bounds are ignored in trait and impl function definitions." [Online]. Available: https://github.com/rust-lang/rust/issues/24011

[131] "Rfc: Where clauses." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0135-where.md

[132] M. Brain, C. Tinelli, P. Ruemmer, and T. Wahl, "An automatable formal semantics for ieee-754 floating-point arithmetic," in *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on*, June 2015, pp. 160–167.

[133] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 2008. [Online]. Available: http://en.wikipedia.org/wiki/IEEE_754-2008;http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

[134] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.38

[135] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180445

[136] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: a modular reusable verifier for object-oriented programs," in *Proceedings of the 4th international conference on Formal Methods for Components and Objects*, ser. FMCO'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 364–387. [Online]. Available: http://dx.doi.org/10.1007/11804192_17

[137] K. R. M. Leino, "Dafny: an automatic program verifier for functional correctness," in *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, ser. LPAR'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 348–370. [Online]. Available: http://dl.acm.org/citation.cfm?id=1939141.1939161

[138] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, ser. SEFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 233–247. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33826-7_16

[139] S. Srivastava, S. Gulwani, and J. S. Foster, "Vs3: Smt solvers for program verification," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 702–708. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_58

[140] M. Kawaguchi, P. Rondon, and R. Jhala, "Type-based data structure verification," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 304–315. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542510

[141] R. Brummayer, F. Lonsing, and A. Biere, "Automated testing and debugging of sat and qbf solvers," in *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 44–57. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_6

[142] R. Brummayer and M. JÄrvisalo, "Testing and debugging techniques for answer set solver development," *Theory Pract. Log. Program.*, vol. 10, no. 4-6, pp. 741–758, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1017/S1471068410000396

[143] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032319

[144] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 93–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_7

[145] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," in *Journal of Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 9, 2015, pp. 53–58.

[146] M. Triska, "Correctness considerations in CLP(FD) systems," Ph.D. dissertation, Vienna University of Technology, 2014.

[147] K. Dewey. [Online]. Available: https://github.com/kyledewey/swipl-z3

[148] T. M. Team, Personal Communication.

[149] "Flawed optimization with bitvector division by zero." [Online]. Available: http://cvc4.cs.nyu.edu/bugs/show_bug.cgi?id=659

[150] "get-model crashes on possible bitvector division by zero." [Online]. Available: http://cvc4.cs.nyu.edu/bugs/show_bug.cgi?id=659

[151] "Assertion violation with bitvector division by zero." [Online]. Available: http://cvc4.cs.nyu.edu/bugs/show_bug.cgi?id=733

[152] "Another assertion violation with bitvector division by zero." [Online]. Available: http://cvc4.cs.nyu.edu/bugs/show_bug.cgi?id=733

[153] "fp.min/fp.max does not work correctly for -0/+0." [Online]. Available: https://github.com/Z3Prover/z3/issues/68

[154] "Bitvector division by zero is possibly overconstrained." [Online]. Available: https://github.com/Z3Prover/z3/issues/190

[155] "Assertion violation involving incremental bitvector division by zero." [Online]. Available: https://github.com/Z3Prover/z3/issues/212

[156] "Assertion violation involving floating point multiplication." [Online]. Available: https://github.com/Z3Prover/z3/issues/215

[157] "Possible bug in fp.sqrt." [Online]. Available: https://github.com/Z3Prover/z3/issues/220

[158] "Soundness / performance bug with fp.sqrt." [Online]. Available: https://github.com/Z3Prover/z3/issues/222

[159] "Memory leak on fp input involving multiple operators." [Online]. Available: https://github.com/Z3Prover/z3/issues/480

[160] "Assertion violation in sat_clause_use_list.h with the theory of floating point." [Online]. Available: https://github.com/Z3Prover/z3/issues/551

[161] "Memory leak in theory of fp with many operations." [Online]. Available: https://github.com/Z3Prover/z3/issues/567

[162] "Memory leak in theory of fp with multiple check-sat commands." [Online]. Available: https://github.com/Z3Prover/z3/issues/615

[163] "Complex memory leak in theory of fp with multiple check-sat commands." [Online]. Available: https://github.com/Z3Prover/z3/issues/643

[164] "Unsat to sat with theory of fp without variables." [Online]. Available: https://github.com/Z3Prover/z3/issues/616

[165] "Incremental solver problem with fp.max." [Online]. Available: https://github.com/Z3Prover/z3/issues/642

[166] T. B. Team, Personal Communication.

[167] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop, "Teaching and learning programming and software engineering via interactive gaming," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1117–1126. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486941

[168] P. Ihantola, "Test data generation for programming exercises with symbolic execution in java pathfinder," in *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, ser. Baltic Sea '06. New York, NY, USA: ACM, 2006, pp. 87–94. [Online]. Available: http://doi.acm.org/10.1145/1315803.1315819

[169] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 41–51. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635912

[170] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, pp. 15–26, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462195

[171] "Moss: A system for detecting software similarity." [Online]. Available: https://theory.stanford.edu/~aiken/moss/

[172] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 76–85. [Online]. Available: http://doi.acm.org/10.1145/872757.872770

[173] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[174] R. A. O'Keefe, *The Craft of Prolog.* Cambridge, MA, USA: MIT Press, 1990.

[175] K. Marriott and H. Sondergaard, "On prolog and the occur check problem," *SIGPLAN Not.*, vol. 24, no. 5, pp. 76–82, May 1989. [Online]. Available: http://doi.acm.org/10.1145/66068.66075

[176] P. Wadler, "Theorems for free!" in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '89. New York, NY, USA: ACM, 1989, pp. 347–359. [Online]. Available: http://doi.acm.org/10.1145/99370.99404

[177] "Gnu prolog reference manual." [Online]. Available: http://www.gprolog.org/manual/gprolog.html

[178] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. pp. 29–60, 1969. [Online]. Available: http://www.jstor.org/stable/1995158

[179] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348 – 375, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000078900144

[180] A. Mycroft and R. A. O'Keefe, "A polymorphic type system for prolog." *Artif. Intell.*, vol. 23, no. 3, pp. 295–307, Jul. 1984. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(84)90017-1

[181] T. L. Lakshman and U. S. Reddy, "Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system," in *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, 1991, pp. 202–217.

[182] Z. Somogyi, F. J. Henderson, and T. C. Conway, "The implementation of mercury, an efficient purely declarative logic programming language," in *IN PROCEEDINGS OF THE AUSTRALIAN COMPUTER SCIENCE CONFERENCE*, 1995, pp. 499–512.

[183] M. Hanus, "Curry: A truly integrated functional logic language," http://www-ps.informatik.uni-kiel.de/currywiki/start.

[184] D. Jeffery, F. Henderson, and Z. Somogyi, "Type classes in mercury," in *ACSC*. IEEE Computer Society, 2000, pp. 128–135.

[185] D. Overton, Z. Somogyi, and P. J. Stuckey, "Constraint-based mode analysis of mercury," in *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '02. New York, NY, USA: ACM, 2002, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/571157.571169

[186] M. Hanus, "A unified computation model for functional and logic programming," in *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, 1997, pp. 80–93.

[187] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the ACM Annual Conference - Volume 2*, ser. ACM '72. New York, NY, USA: ACM, 1972, pp. 717–740. [Online]. Available: http://doi.acm.org/10.1145/800194.805852

[188] R. Ramesh, I. V. Ramakrishnan, and D. S. Warren, "Automata-driven indexing of prolog clauses," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90. New York, NY, USA: ACM, 1990, pp. 281–291. [Online]. Available: http://doi.acm.org/10.1145/96709.96738

[189] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[190] "Implementing cut in tracing meta interpreter prolog." [Online]. Available: https://stackoverflow.com/questions/27235148/implementing-cut-in-tracing-meta-interpreter-prolog

[191] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," Tech. Rep., 2001.

[192] B. Ford, "Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '02. New York, NY, USA: ACM, 2002, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/581478.581483

[193] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi, "Sets and constraint logic programming," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 5, pp. 861–931, Sep. 2000. [Online]. Available: http://doi.acm.org/10.1145/365151.365169